UNIVERSIDAD REY JUAN CARLOS

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

A statistical examination of the properties and evolution of libre software

Doctoral Thesis

Israel Herraiz Tabernero

Ingeniero Industrial

Madrid, 2008

Thesis submitted to the Departamento de Sistemas Telemáticos y Computación in partial fulfillment of the requirements for the degree of Doctor europeus of Philosophy in Computer Science

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Rey Juan Carlos
Madrid, Spain

DOCTORAL THESIS

A statistical examination of the properties and evolution of libre software

Author:
Israel Herraiz Tabernero
Ingeniero Industrial - Industrial Engineer

Codirector:
Jesús M. González Barahona
Doctor Ingenierio de Telecomunicación - Doctor Telecommunication Engineer

Codirector:
Gregorio Robles Martínez
Doctor Ingenierio de Telecomunicación - Doctor Telecommunication Engineer

Madrid, Spain, 2008

October 24th, 2008

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY *Israel Herraiz Tabernero* ENTITLED *A statistical examination of the properties and evolution of libre software* BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF *Doctor Europeus of Philosophy in Computer Science*.

Dr. Jesús M. González Barahona Thesis Codirector	Dr. Gregorio Robles Martínez Thesis Codirector

The committee named to evaluate the Thesis above indicated, made up of the following doctors

Prof. Dr. Manuel Hermenegildo Universidad Politécnica de Madrid Spain President	Prof. Dr. Kevin Crowston Syracuse University USA Member
Dr. Diomidis Spinellis Athens Univ. of Economics and Business, Greece Member	Dr. Tom Mens University of Mons -Hainaut Belgium Member
Dr. Luis López Fernández Universidad Rey Juan Carlos Spain Secretary	

has decided to grant the qualification of

Madrid (Spain), October 24th , 2008.

The secretary of the committee.

A copy of this thesis, and of all the data
sources and tools needed to
replicate this thesis, are available
in the following persistent address

http://purl.org/net/who/iht/phd

More information about empirical research
of libre software may be found at the
Libresoft Research Group homepage

http://libresoft.es

It doesn't matter how beautiful your theory is, it doesn't matter how smart you are.
If it doesn't agree with experiment, it's wrong.

Attributed to Richard P. Feynman

ACKNOWLEDGEMENTS

Four years ago I sent an email to the advisors of this doctoral thesis, because I felt like I wanted to put together two of the passions of my life: to do research, and to write free / open source software.

At that time, I did not know that a single email was going to change my life, to change me, as it has done. It is not only that I have managed to do research while writing some pieces of software, and above all, managed to do so while being paid for it. It is that I have managed to realize that you can really do in your life whatever you convince yourself to do.

How ironic, my mother always told me that when I was a kid. However I never did what I wanted to do, but what I had to do. After getting my master degree I followed that path of duty, and found a good job. How ironic, that good job made me quite unhappy. I refused to accept that life was only that. After so much effort to do the right things, willing to learn whatever I was told to learn, willing to discover the secrets of life, it turned out that the only thing that I found is that following the path indicated by others will not make you happy. Doing the right things is not the right thing to do.

After all, my mother was right. She always told me to follow the path that made happy, to work hard to achieve my dreams, regardless the opinion of other people about those dreams. For that, I am eternally grateful.

I would like also to thank Jesús, who gave me this opportunity (and many others). More than an advisor, he has been a mentor. Every time I face a new problem, I think "what would Jesús do in a situation like this?" I do not want to forget Gregorio. He also has given me (too) many opportunities during these four years, and has been an example about what is working with passion.

This thesis is not my main discovery in these four years. Many (many) people have helped me to discover what love is, and that in the end, as Bill Hicks used to say to finish his shows, life is reduced to a choice between fear and love. The first one who made discover this was my brother, Aitor. His approach to life is an example for me. Thanks to him, now I know that those who love you will always be there, willing to help you, with no reproach. The rest have simply decided to feel fear.

Actually, I did not completely understand that until I met Rocío. I chased her during eleven days from Porto to Santiago, through cornfields and vineyards. Wanting to discover her, I ended discovering more of myself. With her, I finally realized that you can achieve whatever you want, as long as you want it enough, and that after all the truly important things are just a few, and most of us already have them (although we are not aware of it).

I am also very grateful to two special friends: Carlos, who I met in primary school, and Fran, with whom I shared many experiences. Carlos taught me what curiosity is, and Fran what freedom is. After a couple of decades, they have managed to shape their lives as they actually are, and for that, they have set an example to me.

Many other people influenced me during these four years. All my friends at Libresoft with whom I am lucky to work (Teo, Juanjo, Luis, Dani, Carlos, Santi, Roberto A., Roberto C., Juanlu, Gato, Álvaro N., Álvaro del C., Antonio, Miquel, Ernesto, Isabel, Raquel, Liliana, Gato Jr., Paco, Felipe, Verónica, Alicia, Diego and a lot more I am probably forgetting –sorry). All the people I have met in the university. All the people with whom I have shared a flat (Manolo, Juan, José Pablo, Anna, Yanina). All the professors that have supervised me in Milton Keynes (thanks Juan, Andrea), Victoria (thanks Daniel) and Athens (thanks Diomidis); working with them was a real challenge. All the people I have met in Spain and abroad. And last but not least, all the people that I have forgotten to include here. I would include here my complete list of friends in Facebook, but I hate those acknowledgements sections with endless list of names and just a couple of sentences.

Four years of research, of discovery, of studying how software is developed. And after that, rather than how software evolves, I probably have understood better how people evolve.

Thanks to all of you!

Israel Herraiz
September 2008

ABSTRACT

How and why does software evolve? This question has been under study since almost 40 years ago, and it is still a subject of controversy. After many years of empirical research, Meir M. Lehman formulated the *laws of software evolution*, which were a first attempt to characterize the dynamics of the evolution of software systems.

With the raising of the libre (free / open source) software development phenomenon, some cases that do not fulfill those laws have appeared. Are Lehman's laws valid in the case of libre software development? Is it possible to desing an universal theory for software evolution? And if it is, how?

This thesis is a large-scale empirical study that uses a statistical approach to analyze the properties and evolution of libre software. The studied properties are size and complexity. For that study, we have used a set of thousands of software systems, extracted using the packages system of FreeBSD. The evolution study was done using another set of thousands of software projects hosted in SourceForge.net.

With the first set, we measured different size and complexity metrics of the source code of the packages in FreeBSD, and calculated the correlations among the different metrics. We also estimated the distribution function of those properties.

Regarding the second set, we obtained the daily series of number of changes. We applied Time Series Analysis to estimate the kind of process that drives software evolution. We used ARIMA (Auto Regressive Integrated Moving Average) models to forecast evolution.

The results show that a small subset of basic size metrics are enough to characterize a software system. Furthermore, the shape of the distribution of those metrics suggests that the *Random Forest File Model* could be used to simulate the evolution of a software product.

Using Time Series Analysis (TSA), we have found that software evolution is a short memory process. That implies that statistical models of evolution based on TSA are a better option than regression models for forecasting purposes.

Finally, the shape of the distribution of size is the same, regardless of the level of aggregation used to measure it (file, module, software project, etc). That is an evidence of self-similarity in software, and could be an explanation of the fast growth patterns observed in some libre software projects.

Another remarkable contribution of this thesis is that it shows how to perform an empirical study at a large scale, using publicly available data sources. Thanks to this, all the results are repeatable and verifiable by third parties. Therefore, the conclusions of this thesis can be the beginning of a theory of software evolution that is based on empirical findings verified in thousands of software systems.

RESUMEN[1]

¿Cómo evoluciona el software? ¿Y por qué? Esta cuestión ha sido objeto de investigación desde hace casi 40 años, y sigue siendo una pregunta controvertida. Después de varios años de investigación empírica, Meir M. Lehman formuló las *leyes de evolución del software*, que eran una primera aproximación al problema de la caracterización de la evolución del software.

Con la aparición del fenómeno de desarrollo de software libre, se encontraron algunos casos cuya evolución no se regía por esas leyes. ¿Son válidas las leyes de Lehman para el caso del software libre? ¿Es posible obtener una teoría de evolución del software que sea verdaderamente universal? Y si lo es, ¿cómo?

Esta tesis es un estudio empírico realizado a una escala masiva, que emplea un enfoque estadístico para estudiar las propiedades y la evolución del software libre. Hemos seleccionado una muestra de miles de proyectos de software, obtenidos gracias al sistema de paquetes de FreeBSD, con el fin de medir su tamaño y complejidad. Para el caso de la evolución, escogimos otra muestra de miles de proyectos que estaban alojados en SourceForge.net.

Con la primera muestra, medimos diferentes métricas de tamaño y complejidad del código fuente de los paquetes que contiene el sistema FreeBSD, y calculamos las correlaciones entre las diferentes métricas. También estimamos la distribución estadística de esas métricas.

Con la segunda muestra, obtuvimos las series temporales del número diario de cambios. Aplicamos análisis de series temporales para estimar el tipo de proceso que gobierna la evolución del software. Usamos modelos ARIMA para predecir el comportamiento de algunos sistemas de esa muestra.

Los resultados muestran que es posible caracterizar un sistema de software usando métricas básicas de tamaño. Además, la forma de la distribución de tamaño sugiere que se podría emplear el modelo *Random Forest File Model* para simular la evolución del software.

Mediante el empleo de análisis de series temporales, hemos encontrado que la evolución del software es un proceso de memoria corta. Esto implica que para obtener modelos estadísticos de la evolución, es mucho mejor usar modelos ARIMA que modelos de regresión.

Finalmente, la forma de la distribución de tamaño es la misma, cualquiera que sea le nivel de agregación usado para medir tamaño (fichero, paquete, etc). Esto muestra que el software es auto-similar, y podría ser una explicación a los patrones de crecimiento súper-lineales que se han observado en algunos proyectos de software libre.

[1]En el apéndice E se puede encontrar un resumen suficiente en castellano que cumple con los requisitos del artículo 24 del capítulo V de la "Normativa para la Admisión del Proyecto de Tesis y Presentación de la Tesis Doctoral" para las tesis que sean presentadas en otros idiomas diferentes del español, como es el caso de ésta.

Otra contribución de esta tesis es que muestra como realizar un estudio empírico a escala masiva, usando fuentes de datos públicas. Gracias a esto, todos los resultados obtenidos en esta tesis son repetibles y verificables por parte de terceros. Por tanto, las conclusiones de esta tesis pueden ser el comienzo de una teoría de evolución del software que se base en resultados empíricos que se han verificado en miles de proyectos de software.

CONTENTS

1 Motivation **1**

1.1 Introduction . . . 1

1.2 Why libre software? . . . 4

1.2.1 What is libre software? . . . 4

1.2.2 Open software repositories . . . 5

1.3 Overview of this thesis . . . 8

1.3.1 Goals and research questions . . . 8

1.3.2 Main contributions . . . 9

1.3.3 General structure . . . 10

2 State of the art **13**

2.1 Introduction . . . 13

2.2 Software evolution . . . 14

2.2.1 The nature of programs and their evolutionary behavior . . . 15

2.2.2 Law of continuing change . . . 17

2.2.3 Law of increasing entropy/complexity . . . 20

2.2.4 The third law and the notion of feedback . . . 21

2.2.5 The fourth and fifth laws . . . 22

2.2.6 Law of continuing growth . . . 22

2.2.7 Law of declining quality . . . 22

2.2.8 Law of feedback system . . . 23

2.2.9 Principle of software uncertainty . . . 23

2.3 Empirical studies of software evolution . . . 23

2.3.1 Laws of software evolution . . . 24

2.3.2 FEAST and the nineties view . . . 26

2.3.3 Libre software and software evolution . . . 27

2.3.4 The importance of metrics for validation studies . . . 30

2.3.5 Towards large scale investigations . . . 31

2.3.6 The statistical properties of the size of software . . . 33

2.4 Modelling evolution . . . 35

2.4.1 Physical models 36
2.4.2 Statistical models 37
2.4.3 Self-Organized Criticality and the sand pile model 38
2.4.4 The Maintenance Guidance Model 39
2.5 Summary 40

3 Methodology **43**
3.1 Introduction 43
3.2 Correlation analysis 44
3.2.1 Data sources 45
3.2.2 Validation of the sample 46
3.2.3 Selected metrics 48
3.2.4 Statistical analysis 51
3.3 The dynamics of software evolution 53
3.3.1 Data sources 53
3.3.2 Statistical analysis 56
3.4 Forecasting software evolution 59
3.4.1 Data sources 59
3.4.2 Statistical analysis 62

4 Results **65**
4.1 Introduction 65
4.2 First question: Comparability of different metrics 65
4.2.1 Overall correlation 66
4.2.2 Comparability discriminating by package size 67
4.2.3 Comparability discriminating by field of application 68
4.2.4 Summary 69
4.3 Second question: How many metrics to characterize software? 69
4.3.1 Correlation for the overall sample 70
4.3.2 Correlation discriminating by file type 71
4.3.3 Correlation discriminating by package size 72
4.3.4 Correlation discriminating by field of application 76
4.3.5 Analysis of Halstead's level 77
4.3.6 Summary 77
4.4 Third question: The shape of the distribution of the size of software 78
4.4.1 Summary 79
4.5 Fourth question: Self-similarity in software 81
4.5.1 Header and non-header files 81
4.5.2 Shape at more aggregated levels 83
4.5.3 Summary 85
4.6 Fifth question: The dynamics of software evolution 85
4.6.1 Sensitivity analysis 88
4.6.2 Summary 90

4.7 Sixth question: Forecasting software evolution . . . 90
4.7.1 Fitting ARIMA models . . . 91
4.7.2 Accuracy of the models . . . 92
4.7.3 Summary . . . 94

5 Conclusions and further work **97**
5.1 Summary of results . . . 97
5.1.1 Correlation of metrics . . . 97
5.1.2 Software attributes . . . 98
5.1.3 Statistical properties of software . . . 98
5.1.4 Short memory dynamics and ARIMA models . . . 100
5.1.5 Practical implications . . . 100
5.2 Further work . . . 101
5.2.1 The Random Forest File Model applied to the case of software evolution . . . 101
5.2.2 Self-similarity influence on the evolutionary patterns of software . 103
5.2.3 Determinism and evolution . . . 103
5.3 Conclusions . . . 105

A The double Pareto distribution **107**

B Additional results **111**

C Time Series Analysis functions and models **117**
C.1 Autocorrelation Coefficients Function . . . 117
C.2 ARIMA models . . . 118
C.3 Long and short memory processes . . . 118

D Patterns to identify automated files **119**

E Resumen en español **121**
E.1 Antecedentes . . . 121
E.2 Objetivos . . . 122
E.3 Metodología . . . 123
E.3.1 Estudio de las propiedades del software . . . 123
E.3.2 Estudio de la evolución del software . . . 125
E.4 Conclusiones . . . 126

Bibliography **129**

LIST OF FIGURES

1.1 Stages in the life cycle of a traditional software project. The evolution stage starts right after the release of the first operation version of the program. The evolution stage uses to take longer than the development one, and accounts for most of the cost of the software project. 3

1.2 Growth of the release of Debian GNU/Linux: it has doubled its size with each new release. The vertical axis shows the size of the distribution in millions of SLOC. The horizontal axis shows the date of each release. Each bar corresponds to one of the releases of Debian (with the label above the bar). Adapted from [RGBM$^+$08]. 6

2.1 Diagram of the evolution of the laws of software evolution. The laws have increased from three to eight laws. In their latest form, the notion of feedback is the key concept. The term *E*-type has been introduced in the text of the laws, to remark that the laws are only applicable to that kind of software. 24

2.2 Growth in SLOC (lines of code removing blank and comment lines) of the Linux kernel. Each color is one of the branches of Linux. The overall growth follows a quadratic equation. Reproduced from [RAGBH05]. . . . 28

2.3 Growth of smaller, core subsystems of Linux. Some of these subsystems are growing linearly. In general, when looking at the subsystems level, the growth of each subsystem is linear. The sum of many linear segments with different starting points gives the overall quadratic shape. Reproduced from [RAGBH05]. 29

2.4 Number of case studies in empirical studies on software evolution, over the years since the original study by Lehman in 1969. Vertical axis shows the number of case studies, in logarithmic scale. Horizontal axis shows the year of the study. This graph shows only some selected papers, based on the works reviewed in this chapter and in those included in the survey [KCM07]. This thesis (2008) analyzes 12,010 case studies, that were obtained from a previous study [HGBR07b] that analyzed 13,116 case studies 32

3.1 Sample graph that represents the flow of a program. The cyclomatic complexity is the number of regions in the graph. Any graph has at least one region (the surrounding region). So the minimum value of the cyclomatic complexity is 1. In the graph shown in this figure, the value of the cyclomatic complexity is 5. 50

3.2 Breakdown process. The statistical analysis was repeated in subsamples of the overall sample, to test the sensitivity of the results with three different parameters: filetype, package size and field of application. This process helps to determine if the results are due to aggregation effects and therefore are not significant. 52

3.3 Autocorrelation coefficients in a time series. The coefficients are calculated correlating the series against the same series but shifted one position. If the series has n elements, there are $n-1$ coefficients. In the plot, each circle represents a point in the series. The plot shows how the series is progressively shifted one position, and how each coefficient is obtaining by linear correlations of the original series and its shiftings. 57

3.4 Theoretical profiles of the autocorrelation coefficients of long term and short term processes. This diagram shows the mathematical relationship among the coefficients and the time lags. 58

4.1 Scatter plot of SLOC vs Number of modules. (a) Linear scale. (b) Logarithmic scale. The linear trend is clear in the case of logarithmic scale, but not in the case of linear scale. 67

4.2 Correlation coefficient against package size. The distribution of packages was divided in four quantiles, and the correlations were performed using all the packages belonging to each interval. This procedure ensured that the number of points for each correlation was constant. The correlations were made in logarithmic scale, of all the metrics against number of modules. For instance, the circles correspond to the correlation of SLOC against number of modules. The correlation is only significant for large packages, probably because of aggregation effects. 68

4.3 Scatter plot of SLOC against Halstead's level. (a) Linear scale. (b) Logarithmic scale. 71

4.4 Correlation coefficient (SLOC vs McCabe, logarithmic correlation) against package size. The size of the packages was divided in nine intervals (deciles), and the correlation coefficient was calculated using all the files belonging to the packages contained in each decile (this is, across packages). Horizontal axis in logarithmic scale. 73

4.5 Scatter plot of the correlation coefficient of SLOC vs CYCLO for each package, against the size of the package in SLOC. Each point represents a package. We have then $6,556$ points in this plot. The correlation was logarithmic. Although the results are very dispersed, we can observe that most of the points concentrate on values of around $r = 0.80$. Horizontal axis in logarithmic scale. 74
4.6 Smoothing of the correlation coefficient of SLOC vs CYCLO for each package, against the size of the package in SLOC. The correlation was logarithmic. Horizontal axis in logarithmic scale. 75
4.7 Smoothing of the correlation coefficients of SLOC vs all the Halstead's metrics for each package, against the size of the package in SLOC. The correlation was logarithmic. Horizontal axis in logarithmic scale. It seems that the correlation of SLOC vs Halstead's level is not stable with the size of the package. Therefore in spite of the high coefficient when correlated globally, we cannot conclude that there is indeed a correlation between the two variables. 75
4.8 Normality test for the final sample (SLOC metric). Tails deviate from normality, but the main body of the sample is very close to a normal distribution. The test is done in logarithmic scale. 79
4.9 Complementary cumulative distribution function for the SLOC sample. (a) Lower tail (SLOC < 10). (b) Upper tail (SLOC > 3000). The upper tail is a straight line, indicating that the upper values follow a power law distribution. 80
4.10 Complementary cumulative distribution function for the SLOC sample, compared against a lognormal distribution with the same mean and standard deviation, and against a power law distribution with $\alpha = 3.34$. The high tail of the sample deviate from the lognormal, and it is close to a power law distribution (straight line). 80
4.11 Normality test for the logarithm cyclomatic complexity of header files. Low tail deviate from normality and the main body is lognormal as in the case of SLOC. Very high values also deviate from normality. 82
4.12 Complementary cumulative distribution function for the cyclomatic complexity of header files, and comparison against a lognormal with the same mean and standard deviation, and against a power law distribution with $\alpha = 2.44$. The sample is not fitted by a power law neither by a lognormal. 82
4.13 Normality test of the logarithm of package size. The main body of the distribution is normal, and the tails deviate from normality. 83
4.14 Complementary cumulative distribution function of the size of the ports (software packages). The empirical CCDF is compared against a lognormal distribution, with the same mean and standard deviation, and against a power law distribution with $\alpha = 2.97$. The body of the sample matches the lognormal distribution, but the tail is better described by a power law. 84

4.15 Complementary cumulative distribution function of the size of the sections (or fields of applications). The empirical CCDF is compared against a lognormal distribution, with the same mean and standard deviation, and against a power law distribution with $\alpha = 3.14$. The body of the sample matches the lognormal distribution, but the tail is better described by a power law. 84
4.16 Boxplot of the set of Pearson correlation coefficients, for number of commits, and for number of MRs. This graphs shows that most of the cases (surrounded by the main box in the plot) are around high values (~ 0.85) of the coefficient. In other words, most of the projects appear to be short term processes. 87
4.17 Estimation of the probability density function of the set of Pearson correlation coefficients for number of commits and number of MRs. There is a group of projects with coefficients very close to 1, and another group symmetrically distributed around a value of approximately 0.85. Both groups would correspond to short term processes. Long term processes, located in the left tail, are a minority. 87
4.18 Sensitivity analysis. The plots show the values of each one of the properties shown of the projects (vertical axis), compared against the Pearson correlation coefficient for the number of changes series (horizontal axis). Short term projects present values close to 1. The vertical axis of some of the plots are in logarithmic scale. The plots show that there is no pattern when dividing the projects in more homogeneous groups. For all the ranges of the Pearson coefficients, we find projects with values of the properties in a wide range. In the same way, for all the ranges in each one of the properties, we can find projects with a wide range of values of the Pearson coefficients. 89
4.19 Plots of SLOC over time (daily series) for NetBSD, FreeBSD and PostgreSQL. 91
4.20 Autocorrelation (left) and partial autocorrelation (right) coefficients for the first ten lags of the difference of the FreeBSD series (no smoothing applied). 92
4.21 Original series and smoothed series for FreeBSD 93
4.22 Autocorrelation (left) and partial autocorrelation (right) coefficients for the first ten lags of the smoothed FreeBSD series (first difference). 93

5.1 Self similarity in software. The size of source code files, the size of software packages, and the size of sections have all the same statistical distribution. 99
5.2 Events tree. This tree represents all the possible events that may happen in a software evolution project. The top node is the initial situation. In each step, a decision is taken, and that implies to follow a particular path. 104

5.3 Events tree for a short memory process. The density of connections is much higher, and so early decisions do not discard possible future scenarios. 104

A.1 Complementary cumulative distribution function of a double Pareto. The plot shows two straight tails, corresponding to power law distributions with scaling parameter α and β, and a lognormal body that connects the two tails. 109

B.1 Complementary cumulative distribution function for the sample of all (header and non header) files. This plot shows the distribution plots for all the metrics. The legend is the same for all the plots. The plots show a lognormal and a power law fitted to the two extremes of the data. 113
B.2 Complementary cumulative distribution function for the sample of header files. This plot shows the distribution plots for all the metrics. The legend is the same for all the plots. The plots show a lognormal and a power law fitted to the two extremes of the data. 114
B.3 Complementary cumulative distribution function for the sample of non-header files. This plot shows the distribution plots for all the metrics. The legend is the same for all the plots. The plots show a lognormal and a power law fitted to the two extremes of the data. 116

LIST OF TABLES

2.1 Empirical validation of Program Evolution Dynamics in the seventies and eighties. Organization types are: Computer Manufacturer (CM), Financial Institution (FI) and Software House (SH). System types are: Operating System (OS), Banking System (BS), Transaction OS (TOS), Applications (APP) and Stock Control (SC). Size given in number of modules. Reproduced from [Pir88]. 15
2.2 Laws of software evolution in 1974 (called laws of Program Evolution Dynamics by that time). These laws were based only on the empirical findings using OS/360 as case study. Reference: [Leh85b], which is a reprinted edition of [Leh74]. 16
2.3 Laws of software evolution in 1978 (called laws of Program Evolution by that time). Reference: [Leh85c], which is a reprinted edition of [Leh78]. . . 17
2.4 The last three laws were changed in 1980, that remained unchanged in the book published in 1985 [LB85]. Reference: [Leh80]. 18
2.5 Laws of software evolution, reformulated in 1996 [Leh96] and empirically revalidated in 1997 [LRW+97]. This form remained in the book published in 2006 [MFRP06]. This can be considered the current formulation of the set of the Lehman's laws. 19
2.6 Analogy between the sand pile model and the evolution of a software system. Reproduced from [Wu06]. 39
2.7 Classification of empirical studies on software evolution 42

3.1 Summary of the validation process of the sample 47
3.2 Selected metrics to measure size and complexity. 51
3.3 Statistical properties of the sample of 3821 projects. # Devs. are number of developers. SF.net age indicates the number of months that the project has existed in SF.net. CVS age indicates the number of months that have elapsed between the first and the last commits in the CVS repository. SLOC measures size in Source Lines of Code (it excludes blank and comment lines). #MRs is number of modification requests. 56

3.4 Criteria to select the values of p and q in an ARIMA model. ACF means autocorrelation function. PACF means partial autocorrelation function. Reproduced from [SS06]. 62

4.1 Coefficient of correlation of the number of modules against the rest of metrics. Correlations performed in logarithmic scale. 66

4.2 Summary of the correlation coefficients calculated for all the fields of application. All the correlations were made against the number of modules and in logarithmic scale. The last column is the overall coefficient calculated for all the packages as a single sample. 69

4.3 Descriptive statistics of the sample of files. The table shows the mean, standard deviation and quantiles (minimum, first, median, third and maximum) for the different metrics using all the files of the sample. 70

4.4 Correlation among the logarithm of all the metrics. This sample includes both header and non-header files. 72

4.5 Correlation coefficients of SLOC against the rest of complexity metrics for the header and non-header files. 72

4.6 Summary of correlation coefficients calculated by field of application. Correlations against SLOC. The last column shows the correlation coefficient computed for the whole sample, without any classification (by filetype, by package size, etc). 76

4.7 Statistical properties of the set of Pearson correlation coefficients. The values range from very low (~ 0.3, corresponding to long term processes) to very high (~ 0.99, corresponding to short term processes). With only this information, we cannot quantify how many projects could be classified in each category (short or long term). 86

4.8 Quantiles (in %) of the sample of Pearson correlation coefficients, for number of commits, and number of MRs. Less than 40% of the projects present a value lower than ~ 0.8. 88

4.9 Mean squared relative errors (MSRE) and standard deviation of the squared relative errors (Sd. dev.) for the time series and regression models. 94

B.1 Parameters of the double Pareto distribution, for all the metrics using the overall sample. The first column shows the mean of the lognormal part ($\bar{x}$). The second column shows the standard deviation of the lognormal part (s). The third column shows the value of the transition point from lognormal to power law (x_{min}). The fourth column shows the value of the power law exponent (α). 112

B.2 Parameters of the double Pareto distribution, for all the metrics using the sample of header files. The first column shows the mean of the lognormal part ($\bar{x}$). The second column shows the standard deviation of the lognormal part (s). The third column shows the value of the transition point from lognormal to power law (x_{min}). The fourth column shows the value of the power law exponent (α). 112
B.3 Parameters of the double Pareto distribution, for all the metrics using the sample of non-header files. The first column shows the mean of the lognormal part ($\bar{x}$). The second column shows the standard deviation of the lognormal part (s). The third column shows the value of the transition point from lognormal to power law (x_{min}). The fourth column shows the value of the power law exponent (α). 115

CHAPTER
ONE
Motivation

How and why does software evolve? This question has been under study since almost 40 years ago, and it is still a subject of controversy. After many years of empirical research, Meir M. Lehman formulated the *laws of software evolution*, which were a first attempt to characterize the dynamics of the evolution of software systems.

With the raising of the libre (free / open source) software development phenomenon, some cases that do not fulfill those laws have appeared. Are Lehman's laws still valid in the case of libre software development? Is it possible to design an universal theory for software evolution? And if it is, how?

This chapter analyzes the goals of software evolution as a field of research, and how the increasing availability of libre software repositories can help to achieve those goals. Finally, this chapter presents an overview of this thesis, and of its contributions.

1.1 Introduction

Computing is one the most sophisticated technologies that the human being has created. A computer can do any task that can be expressed as an algorithm. Although this may have some limitations (some believe that many tasks cannot be expressed as algorithms [Pen99]), computing and software development have provided the tools that nowadays are used to solve many of our problems.

Together with the raising of computing and software development, the discipline of *Software Engineering* was born. In 1968, the NATO Scientific Committee organized the first conference on Software Engineering [NR69]. This new discipline aimed (and still aims) to provide the techniques to build software in an effective way, in an environment where computers were gaining more importance. Just like any other engineering, Software Engineering tries to define the standard practices, methods and data necessary to build software (that was for instance the exact topic of one the working papers presented in that conference [Ran69]).

The discipline has evolved, and nowadays it is a recognized field. There are university degrees in the topic, both at the Master and PhD level; even a *body of knowledge* has been defined [BDA$^+$99]. Many books explain how software must be built, and how programming teams must be organized to accomplish their duties in the most effective way (for instance [Som06]).

Interestingly, in the last years a different way of building software has emerged: libre (free / open source) software[1]. Libre software is usually developed in communities, distributed all over the world, communicating through the Internet. In many cases, the members of the development teams do not even have met in person. Those people come from different cultures, speak different languages, work in different time zones, do not write requirements but take them as implicit, and yet they manage to deliver quality products that are comparable to those developed in industrial environments.

Meir M. Lehman is the pioneer in software evolution studies. He identified the phenomenon as early as 1969 in an internal report, while working for IBM. The whole story on how software evolution was identified may be found in the seminal book on software evolution [LB85], published in 1985. In that book (as well as in some other previous publications), the Lehman's *laws of software evolution* were formulated. These laws were one of the first steps towards a theory for software evolution. They have been reformulated to keep them updated to the new software development environments [Leh74, Leh78, Leh96].

Software evolution refers to the phenomenon of software change and growth. The environment where software has to work changes over time, and software itself must adapt to this changing environment. Of course, software does not change by itself, but because of the action of development and maintenance teams. In traditional environments, software evolution is considered to start right after the first operational release. Hence the life cycle of software has two main stages: development and evolution (see figure 1.1). Most of the effort and costs are usually devoted to the evolution stage.

In the case of libre software, there is not clear distinction between two stages in a project. In those environments, software is usually released early with a minimal functionality, to gain attention and to build a community around the project to drive it. It is the principle known as *release early, release often* [Ray98]. According to this principle, libre software is released as soon as it reaches some minimal functionality, and new versions are released often. This cycle is driven by a software community, that collaborates in the maintenance of the software products, and in the development of new functionality providing feedback.

Therefore, the evolutionary pattern is different to the classical scheme described by Lehman. In libre software,development and evolution happen at the same time. This shift in the evolutionary behavior might explain why some libre software projects appear to be evolving in a different way to the Lehman's predictions. This controversial

[1]Thorough this thesis I will use the term *libre software* to refer both to *free software* (as defined by the Free Software Foundation) and *open source software* (as defined by the Open Source Initiative). The term is reviewed in section 1.2.1

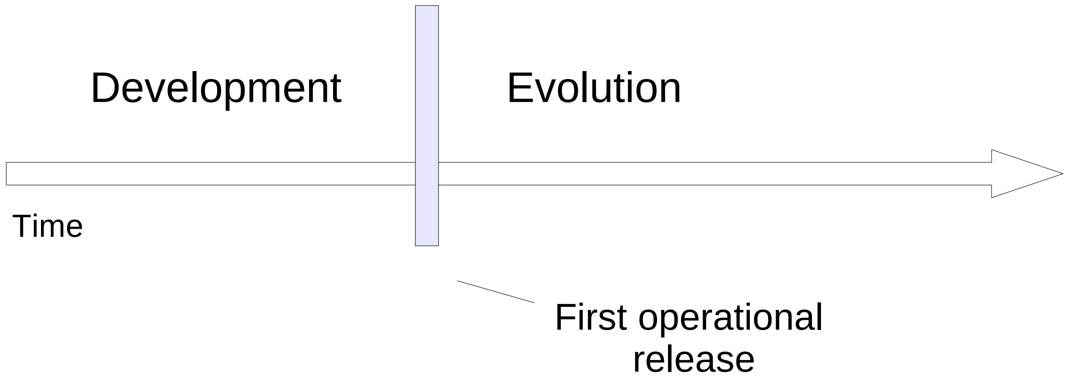

Figure 1.1: Stages in the life cycle of a traditional software project. The evolution stage starts right after the release of the first operation version of the program. The evolution stage uses to take longer than the development one, and accounts for most of the cost of the software project.

topic has been subject of research in recent years. Godfrey and Tu [GT00] highlighted that the Linux kernel (probably the most famous libre software project) was growing at a *superlinear* rate. This result was against Lehman's predictions, that state that software grows at a declining rate because complexity increases with time. That work raised the question of whether or not Lehman's laws were valid for the case of libre software.

Around the validity of the Lehman's laws for libre software evolution, many other questions have appeared. One of these questions is about the metrics used to study evolution. Are studies that use different metrics comparable? How many and which metrics can be used to characterize software evolution? In the case of the works mentioned above, the work by Godfrey and Tu used different metrics that the original works by Lehman. Can we conclude that the laws of software evolution are not valid for Linux, using different metrics?

Another question is about modelling software evolution. In the case of libre software, it is usually developed in communities, where many different actors interact, with complex patterns. In a typical libre software project, we can find users, casual contributors, defect reporters, developers, etc. Most of them behave according to their own interests. In those communities, project management is usually distributed, and decisions are difficult to impose. There are not hierarchies, or at least no an only hierarchy accepted by all the actors. Therefore, developers (or any other kind of actor in general) can not be compelled to do particular tasks, even if those tasks are urgent. Furthermore, there is a lack of written requirements, and in general, very few documentation about the different processes that take part in the community.

However, despite this apparent lack of planning, these communities achieve to develop quality products that fulfill the demands of users. Forecasting the evolution of these projects can be useful for management purposes. Not only for the community itself, but also for third parties like companies interested in getting involved in the community. Unfortunately, under such scenario of complex interactions, forecasting and modelling can be a risky business.

The study of the dynamics of software evolution, this is, of the mechanisms that drive software evolution, is one of the approaches to reduce the risk inherent to forecasting and modelling software evolution. Which are the principles that drive software evolution? Lehman attempted to summarize those principles in the form of the laws of software evolution. However, as previously mentioned, those laws seem not to be valid in the case of libre software. Which are the principles and processes that generate those software products? If we determine the kind of dynamics of software evolution, we could obtain better models for forecasting and managing libre software projects.

Finally, there is an additional question regarding software complexity. The laws of software evolution predict that the growth rate of a project must decrease because complexity increases with time. This makes it more difficult to make changes, and the resulting effect is a slower growth. However, in the previously mentioned case [GT00], the growth rate increased with time. Is the complexity of these conflicting cases growing? Are the metrics used by Lehman actually measuring complexity?

1.2 Why libre software?

Libre software is becoming mainstream. It is attracting an increasing attention in the research field, and it is gaining user share. The number of projects and the size of code written under the scope of libre software projects have been rapidly increasing during the nineties and the first part of twenty-first century. As an example, see figure 1.2, that shows the growth in number of packages of the different releases of Debian. A side effect of this increasing activity in the libre software community is that a large amount of data and software repositories have been made available for research purposes.

1.2.1 What is libre software?

In this thesis I use the term *libre software* to refer both to what is commonly known as *free software* and *open source*. A software is said to be libre (free / open source) if it fulfills the following four requirements [Sta02]:

- The freedom to run the program, for any purpose.
- The freedom to study how the program works, and adapt it to your needs. Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor.
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits. Access to the source code is a precondition for this.

The term *free software* was originally coined in 1984 by Richard M. Stallman [Sta02]. In the English language, *free* has two meanings: free as in freedom and free as gratis.

This led to some confusion, because it was usual to think of free software as merely software that costed nothing. As shown above, free software does not refer to price, and it can be free of charge or not.

Some people thought that the association between "free software" and "software free of charge" made it difficult to spread free software within companies. In 1998, those people created a new term: *open source* [Per99]. It tried to emphasize the availability of the source code, which is something that most of the companies would find attractive. There has been a lot of discussion around the two terms. For instance, Richard Stallman has repeatedly stated that *free software* is a better name than *open source* [Sta02].

Some other terms are lately becoming popular within the research community: FOSS (sometimes written as F/OSS) and FLOSS. FOSS is an acronym for Free and Open Source Software. FLOSS is another acronym, this time for Free, Libre and Open Source Software.

The term *libre software* originated in Europe. Gonzalez-Barahona makes a deep review of the origin of the term [GB04]. This term has been used in some research projects funded by the European Comission, such as CALIBRE[2]. However, in the last years, it seems that the popularity of other terms like FLOSS is increasing.

Although nowadays all the terms and acronyms mentioned above are becoming popular, I consider that the term *libre software* gives a clearer idea of what is free and open source software. I use the term for that reason, and also to avoid the exclusive use of *free software* or *open source software* terms.

1.2.2 Open software repositories

Nowadays, the number of libre software projects may be estimated in the order of $100,000$. This number has been mostly reached in a time span of about ten years. We can use the growth of the Debian GNU/Linux distribution as an estimation of the overall growth of libre software. The size and growth of Debian has been studied and reported [RGBM$^+$08]. Figure 1.2 shows the sizes of the releases of Debian over time. The vertical axis shows the size in *Source Lines of Code* (basically, it is lines of code removing blank lines and comment lines, see section 3.2.3 for a definition of this metric). The horizontal axis corresponds to the date when each version of Debian was released. Each bar corresponds to a release of Debian. The plot includes only the most modern versions of Debian because the packages corresponding to earlier versions could not be obtained.

We can observe that during the last years of the nineties (release 2.0) and the first years of the twenty first century (release 2.2), Debian doubled its size. It doubled again between the releases 2.2 and 3.0, and again between 3.0 and 3.1. This means that, during nine years, the size of the software included in Debian has doubled approximately every three years. If we take Debian as a proxy of the overall community, that means that the libre software that was being developed in those years was doubling every three years.

[2]http://calibre.ie

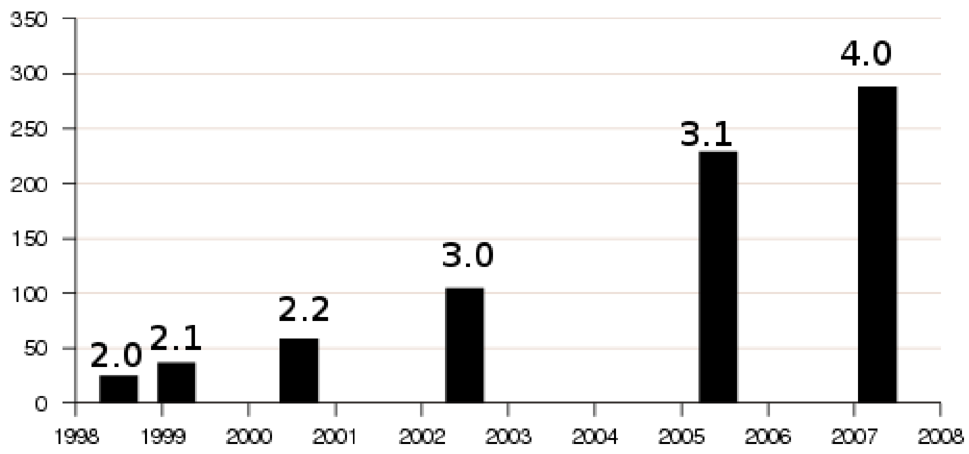

Figure 1.2: *Growth of the release of Debian GNU/Linux: it has doubled its size with each new release. The vertical axis shows the size of the distribution in millions of SLOC. The horizontal axis shows the date of each release. Each bar corresponds to one of the releases of Debian (with the label above the bar). Adapted from [RGBM$^+$08].*

As said before, libre software projects are usually developed in communities, using several kinds of repositories to support the development process. The fast growth of the libre software community has produced a large set of those repositories that can be used for research purposes. A comprehensive review of the repositories and the kind of data that are usually available in libre software projects can be found in the Gregorio Robles' PhD thesis [Rob06]. Summarizing, the main kinds of repositories are the following:

- **Source code releases**

  Libre software projects offer their products as software releases, usually in source code form. Those releases include all the source code necessary to build the system, as well as user documentation, and sometimes documentation intended for development.

  Distributions package that source code. They sometimes also apply patches. Therefore when studying source code that comes from distributions, it may differ from the original code written by the project.

- **Source code management systems**
  In libre software projects, many people work at the same time in the same code base. Source code management (SCM) systems are used to coordinate the work between all the actors in the community.

  SCM systems track all the changes (as well as some meta-information, like the author of the change, the date and time, etc) and make it possible to merge changes to the same entity coming from different people.

From a research point of view, using the control version system a researcher can retrieve the code base as it was at any moment in the history of the project. Furthermore, all the meta-information can be used to study the development process. The meta-information makes it possible to know, for every change in the code base, who made the change, where, when, and probably why (because all the changes are accompanied with a little text message explaining the change).

- **Issue tracking systems**
  Issue tracking systems allow users to inform of the defects that they find, and to request enhancements. If they have technical skills, and given that the source code is available, they can provide solutions to the defects in form of patches.

  Developers also participate in this system, both reporting bugs and fixing them. Both users and developers can discuss about all the reported issues, because each bug report can have comments attached to it.

- **Mailing list archives**
  Mailing lists are the main communication channel used in libre software communities. Lists are usually organized by intended audience. At least, it is usually to find a list for developers and another one for users.

  Anyone may subscribe to a mailing list to receive messages sent to the list. Some lists require prior subscription to be able of sending messages. All the messages that are sent to the lists are stored in archives, that contain the complete history of the list. Sometimes, a subscription to the list may be required to access them. Those archives make it possible to track all the public communications among developers and users, to study the development process.

- **Other communication tools**
  The above mentioned repositories are the most usual, and the most used for research purposes as well. However, several other types of repositories and communications may be found in a community.

  For instance, logs of Internet Relay Chat (IRC) channels, where users and developers communicate in live discussions (contrarily to mailing lists, that are asynchronous), or forums, usually intended only for users and not for developers.

  Other source suitable for research is package information, which includes for example information regarding the dependencies required to build or run each package.

All the sources mentioned above were intended for communication and development purposes. But all the information is publicly available for anyone. This makes it possible to use that information for research purposes. Considering the size of the population of libre software projects (in the order of $100,000$), if empirical studies can be automated, the results could be obtained for a large set of case studies.

In other words, the statistical approach originally proposed by Lehman [Leh74] is now possible thanks to the popularization and the growth of the libre software community. Perhaps because of this fact, the discipline of Mining Software Repositories (MSR) has gained importance in the last years. This discipline empirically studies the information that software repositories contain, to extract knowledge that can be applied to improve the software development process.

Two examples that take advantage of the availability of thousands of open software repositories available for research are the FLOSSMole and the FLOSSMetrics projects. FLOSSMole is gathering information about all the projects hosted in well-known *forges* (web-based integrated development environments that use to offer hosting for libre software projects free of charge). FLOSSMetrics is gathering metrics for thousands of libre software projects at finer grained levels; those metrics are obtained directly from the repositories of the projects.

1.3 Overview of this thesis

Software evolution, a mature discipline, has faced the appearance of a new way of organizing software development teams, and of a new culture of software development in distributed environments where software is shipped as soon as possible, even if it is not yet ready for users.

The original intention of Lehman was (and still is) to design a theory of software evolution. As I will show in section 2.2.4, Lehman *predicted* one of the approaches that can be used for the quest of such theory: statistics and time series analysis. Fortunately, the raising of libre software has made available thousands of repositories to study the evolutionary behavior of software. Indeed, in this thesis I study the static properties of two samples of $12,010$ and $6,556$ libre software projects and the evolutionary profile of another $3,821$ libre software projects, using a statistical approach. The main goal is to address the questions previously mentioned in this chapter.

1.3.1 Goals and research questions

The main goal of this thesis is the following:

> To study the evolution and properties of a statistically significant amount of libre software projects, by means of empirical studies and using a statistical approach, to determine whether or not some commonalities do exist, that could be used to formulate a universal theory of software evolution.

This thesis is based on repeatable observations about the properties of software and about the dynamics of software evolution, which could eventually be transformed into laws, that could be explained by a hypothetical theory of software evolution. In other words, it studies the evolution and properties of software, using an empirical approach and statistical techniques, to extract patterns and commonalities in a statistically significant set of case studies.

Besides this main topic, thanks to the methodology and the information contained in the datasets studied in this thesis, I will answer the following questions as well:

1. Are the laws of software evolution valid for the case of libre software?
   Some studies [GT00, RAGBH05] seem to be in conflict with the predictions of the laws of software evolution [LB85]. But those studies use different metrics than the original studies by Lehman. In this thesis I use two approaches to answer to this research question: first I analyze the works done by Lehman during the seventies and eighties (most of them republished in a book in 1985 [LB85]), and secondly I use an empirical approach to determine whether different size and complexity metrics are comparable or not.

2. Linking with the previous question, I solve the question of which metrics should be used to characterize the size and complexity of a software system (bounded to the case of the C programming language).

3. Next question is about the statistical distribution of software size and complexity. The shape of the statistical distribution of size and complexity can give information about the kind of generative processes that drive software evolution. That information can be used to design models of software evolution.

4. Linked with the above question, I will determine whether or not the statistical distribution of size changes with the scale of the measurements. This is, whether the distribution is the same measuring the size of software at different levels of granularity (file, module, project). That also provides information about the generative processes that drive software evolution.

5. Next question is about the dynamics of software evolution. I have used time series analysis to analyze which kind of dynamics is driving software evolution. In particular, to determine whether it is a long or short range correlated process.

6. Linked with the previous question, I will show how to use that information to obtain accurate models of the evolution of a software product.

1.3.2 Main contributions

The main contributions of this thesis can be summarized as follows:

- It is the first empirical study performed on a large dataset: two samples of $12,010$ and $6,556$ software projects for the properties of source code, and another one of $3,821$ projects for the time series analysis. This makes it possible for the first time to use the statistical approach suggested by Lehman in 1974 [Leh74].

- It shows that different size metrics are highly correlated. Therefore, studies using those metrics are comparable. In particular, all the studies using SLOC are comparable with the classical studies done by Lehman [LB85].
- For the case of the C programming language, it shows that traditional complexity metrics are highly correlated with very simple size metrics (like lines of code).
- It shows that the statistical distribution of software size is a double Pareto. The theoretical models of processes that generate that kind of distributions can be adapted to simulate software evolution.
- It finds evidences of self-similarity in software. The shape of the statistical distribution of software size is the same regardless of the level of granularity used to measure size (file, module, project).
- It uses time series analysis to determine what kind of dynamics (short or long memory) is driving the evolution of software. The finding of a short memory dynamics made it possible to successfully apply ARIMA (Auto-Regressive, Integrated, Moving-Average) models to forecast the evolution of software projects. My proposal of using an ARIMA model won the 2007 MSR Challenge on predicting the evolution of Eclipse [HGBR07a].

1.3.3 General structure

This first chapter has shown the main research questions addressed in this thesis. Most of them arise from the conflict between the empirical studies of the evolution of libre sofwtare [GT00, RAGBH05] and the classical works by Lehman [LB85]. Those works are reviewed in chapter 2, that describes the state of the art. Besides the mentioned works, it reviews the empirical studies done in the scope of software evolution modelling. It also reviews the scale of empirical studies since the first studies in the sixties and seventies. Because of the availability of libre software repositories, this scale has been increasing in the last years.

After reviewing the state of the art, chapter 3 shows the methodology followed to obtain the results of this thesis. I have performed three different studies: correlation analysis of size and complexity metrics, empirical study of the dynamics of software evolution using time series analysis, and statistical modelling of evolution, also using time series analysis. For the three studies, the methodology can be divided in two stages: data collection and statistical analysis. Some details (like the full equations for the double Pareto distribution) about the statistical analysis are shown in the appendixes

The next chapter includes the results using the procedures described in the previous chapter. However, given the size of the samples used in this thesis, some additional results are shown in the appendixes.

Finally, chapter 5 includes the conclusions and further work of this thesis. Because of the correlation found in the previous chapter, I recommend to use only SLOC to study the evolution of software written in C language. I have also found that the distribution of software size is a double Pareto. That implies that the generative process for software evolution could be described using the Random Forest File Model [Mit04b]. As further work, I recommend to use that model to simulate the evolution of software systems. That distribution appears at different scales, which is in evidence of self-similarity. Regarding the dynamics, as shown above, it is a short range correlated process.

CHAPTER
TWO
State of the art

2.1 Introduction

This thesis is fundamentally an empirical study of the properties of software and its evolution, within the realm of libre software. In this chapter, I review the empirical research about software evolution. I classify the works in three main categories: laws of software evolution, empirical studies on evolution, and modelling studies. In the first category, I include the classical works done by Lehman and colleagues that led to the publication of the seminal book on software evolution in 1985. In the second category, I include all the empirical studies that have tried to validate (or invalidate) those laws. Finally, in the third category I include those studies that have tried to obtain a model for the evolution of software by using different approaches, that I have labeled as *statistical*, *physical* and *blended*. Statistical models cannot explain why software evolves, but can help to manage, forecast and control a software system. Physical models, in the other hand, try to explain why software evolves, but to date no physical model has been proved to be accurate enough when tested against real data. In the last years some authors have tried a blended (both statistical and physical) approach: based on the statistical properties of software, some models that explain why software evolves have been derived.

This chapter also reviews the evolution of studies themselves. This is, the topic and scale of research works in the field have changed over the years. For example, recent studies are done at a larger scale than early studies.

Finally, I also review works about properties of software. During the years, larger datasets have become available, and some authors have tried a *survey* approach to characterize properties of software. This resembles the work done by Knuth in 1971 [Knu71]. By that time, FORTRAN was a very popular programming language. Different FORTRAN compilers were available in the market, which were supposed to be designed for the *most typical case*. However, no one had studied which was the most typical case. Knuth made a survey of FORTRAN programs, and studied the properties of the source code (basically, the kind of statements used, and the frequency of each statement). By analyzing "what programmers really do" at a large scale, he found out which statements and structures were the most common. With that information, the design of compilers could be improved. Following a similar approach, the studies presented here are focused in the distribution of size and complexity in software, and in the implications that those distributions have for software evolution modelling.

The rest of the chapter is as follows. Section 2.2 is a thorough review of the pioneering work by Lehman *et al.*, that supposed the first steps of software evolution as a field of research. Section 2.3 reviews some empirical studies that have tried to validate (or invalidate) the laws of software evolution. This section also reviews some other empirical works, that although not having the validation of the laws as main goal, provide some insightful conclusions for the validation of the laws (for instance, whether studies that use different metrics may be compared or not). It also reviews how they have changed over time, and how their scale have increased thanks to the availability of large and public software datasets. Section 2.4 reviews the papers that try to obtain a model for software evolution, using the three approaches previously mentioned. Finally, the last section includes a classification of the main studies presented in the chapter, and some conclusions about how their relationship with this thesis.

2.2 Software evolution

Meir M. Lehman is the pioneer in software evolution studies. In 1969, he performed an empirical study within IBM, in order to improve the programming effectiveness of the company. Although the internal report (initially confidential, but later published as [Leh85a]) had no impact, it was one of the first empirical studies on software development, and a pioneer work on software evolution.

The findings of that work led to the stating of the principles of *Program Evolution Dynamics*, as it was called at that time. The first study was only based on the case of the OS/360 operating system. In the second International Conference on Software Engineering, in 1976, a new paper with more empirical support for those principles was presented [LP76] (republished as [LP85]). In the late seventies and early eighties, the new discipline of *Program Evolution Dynamics* and its principles were empirically validated with other software projects (see table 2.1).

Those works led to the publication of the seminal book on software evolution [LB85] (although the field was not yet called *software evolution* at that time). However, the laws of software evolution were stated more than 10 years before the publication of that book. Before entering into details, it is interesting to remark that the laws have evolved themselves. They were first stated in 1974, as part of the inaugural lecture of Lehman as professor of the Imperical College [Leh74] (republished as [Leh85b]). The text of the laws at that time is shown in table 2.2. By that time, the laws were based only in the empirical findings using OS/360 as case study. Later, in 1978, two more laws were added (see table 2.3). Both are boundary conditions for the third law. The last three laws

System	Type	Org.	Period of analysis	Size	Releases	Users
OS/360	OS	CM	1966–1975	1152–5300	21	Very many
DOS	OS	CM	8 years	438–2142	31	Very many
EXECUTIVE	BS	FI	1973–1978	657–967	12	2 sites
OMEGA	TOS	CM	1972–1974	335–388	9	Many
BD	APP	FI	1973–1977	42–66	—	Few
CCSS	SC	SH	1972–1979	971–1483	58	Few
VME/B	OS	CM	1975–1982	1728–3239	10	Many

Table 2.1: *Empirical validation of Program Evolution Dynamics in the seventies and eighties. Organization types are: Computer Manufacturer (CM), Financial Institution (FI) and Software House (SH). System types are: Operating System (OS), Banking System (BS), Transaction OS (TOS), Applications (APP) and Stock Control (SC). Size given in number of modules. Reproduced from [Pir88].*

were modified in 1980 (see table 2.4), and finally, in 1996, three more laws were added (see table 2.5). This time, the concept of *E*-type software was introduced in the text of the laws (although the concept itself dates back to 1980 [Leh80]). That concept is part of the *SPE* classification scheme that Lehman proposed for the kinds of software to which his laws could be applied. Until the proposal of this scheme, the laws were supposed to be valid for *large* programs. However, Lehman found some difficulties in empirically defining if a particular program was large or non-large, and decided to further elaborate the concept.

2.2.1 The nature of programs and their evolutionary behavior

When the laws of software evolution were first published, Lehman argued that they were only applicable to large programs. Many of the facts contained in the laws were consequences of having large programming teams, with more than one managerial level, with some history that could be studied to test the laws, and with a large base of users that could provide feedback to the programming process.

The *physical* size of the code (for example, the number of lines of code) can be a proxy measure for the *largeness* of the software system. Large programming teams are supposed to produce large systems. Or, from another point of view, one does not need a large team to produce a small (in the physical sense) system. However, it was not clear which values could be considered as large and which ones could not.

For this reason, Lehman decided to change the field of applicability of his laws, and formulated the *SPE* scheme to classify programs [Leh80].

In his original conception, the scheme defined three classes of programs:

I	*Law of continuing change* A system that is used undergoes continuing change until it becomes more economical to replace it by a new or restructured system.
II	*Law of increasing entropy* The entropy of a system increases with time unless specific work is executed to maintain or reduce it.
III	*Law of statistically smooth growth* Growth trend measures of global system attributes may appear stochastic locally in time and space but are self-regulating and statistically smooth.

Table 2.2: *Laws of software evolution in 1974 (called laws of Program Evolution Dynamics by that time). These laws were based only on the empirical findings using OS/360 as case study. Reference: [Leh85b], which is a reprinted edition of [Leh74].*

- *S*-type programs are derivable from a specification, and can be formally proven correct or not.
- *P*-type programs attempt to solve problems that can be formally formulated, but that are not affordable from a computational point of view. Therefore the program must be based on heuristics or approximations to the theoretical problem.
- *E*-type programs are reflections of human processes. This kind of programs attempt to solve an activity that involves people.

The evolution phenomenon refers to *E*-type software. Lehman explicitly introduced the term in the latest form of the laws of software evolution (see table 2.5 on page 19).

The *E* stands for *evolutionary*. This kind of software is a model of human activity and organization. Because the ways people interact and behave change, this type of software has to change to stay synchronized with the new environment. Section 2.2.2 shows in more detail this evolutionary process.

Unlike *E*-type, *S*-type software does not show an evolutionary behavior. Once the program is written, it is either correct or not. If it is not correct, it will not be released and therefore will not reach the evolution stage. However, if it is correct, it is finished, never reaching the evolution stage. Because the specification of this type of programs is a mathematical concept, the problem that the program solves will remain the same regardless of changes in the environment.

The original definition of *P*-type was a little ambiguous: programs that mechanize a human or societal activity. Many times, programs that appeared to be *P*-type showed characteristics proper of either *E*-type or *S*-type software. Indeed, Lehman himself stopped using the *P* category. To overcome this ambiguity, in 2006 a new scheme was proposed, under the name *SPE+* [CHLW06]. This time, the *P*-type software referred to

I	*Law of continuing change* A program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost effective to replace the system with a recreated version.
II	*Law of increasing complexity* As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.
III	*Law of statistically regular growth* Measures of global project and system attributes are cyclically self-regulating with statistically determinable trends and variances.
IV	*Law of invariant work rate* The global activity rate in a large programming project is invariant.
V	*Law of incremented growth limit* For reliable, planned evolution, a large program undergoing change must me made available for regular user execution (released) at maximum intervals determined by its net growth. That is, the system develops a characteristic average increment of safe growth which, if exceeded, causes quality and usage problems, with time and cost over-runs.

Table 2.3: *Laws of software evolution in 1978 (called laws of Program Evolution by that time). Reference: [Leh85c], which is a reprinted edition of [Leh78].*

paradigm based software. In essence, this kind of software is similar to *S*-type software. The only difference is that in this case user satisfaction depends on the system maintaining consistency with a single paradigm over the program lifetime. But in essence, the concept has remained the same, and the definition given above still applies in the scope of the new *SPE+* scheme.

Summarizing, only *E*-type software shows an evolutionary behavior. This thesis will deal with *E*-type software, and the rest of discussions and considerations are always referred to this type of software.

2.2.2 Law of continuing change

The first law is the basic principle of software evolution. Why software evolves? Software is not an isolated agent. It works in an environment that changes. In order to keep software coupled to its environment, it needs to change too. From other point of

III	*The Fundamental Law of Program Evolution* Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and variances.
IV	*Conservation of Organizational Stability (Invariant Work Rate)* During the active life of a program the global activity rate in the associated programming project is statistically invariant.
V	*Conservation of Familiarity (Perceived Complexity)* During the active life of a program the release content (changes, additions, deletions) of the successive releases of an evolving program is statically invariant.

Table 2.4: *The last three laws were changed in 1980, that remained unchanged in the book published in 1985 [LB85]. Reference: [Leh80].*

view, software is just a model of some portion of the reality. As that reality changes, software has to change too. Otherwise it will become useless after some time. This law has persisted in the later revisions of the laws (see tables 2.2, 2.3 and 2.5).

Although this law could seem straightforward, it is a basic principle of software development and evolution. Software is a tool intended to serve humans. Although there is software that interacts directly with other software rather than with humans, in the end, all the software is intended to interact with and help humans[1]. In other words, software is just a reflection of human processes in the technical sphere (or a model of reality, to use the same terms than above). People communicate, write documents, listen to music, work together in organizations, etc. And software helps and serves people to do those tasks.

People change. Human processes change. And software, which is a reflection of those processes, must be changed to keep it synchronized with the reality that it is modelling. Summarizing, the first law of software evolution is a fundamental principle inherent to software. It has remained in the same form since its original publication in 1974[2], which evidences its robustness (all the other have changed since their original conception).

[1]For clarification, we insist that the discussion on the laws refers only to the concept *E*-type software presented in the previous section

[2]Although in its latest form, the concept of *E*-type software has been introduced in the text of the law.

I	*Law of Continuing Change* An E-type system must be continually adapted, else it becomes progressively less satisfactory in use
II	*Law of Increasing Complexity* As an E-type is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.
III	*Law of Self Regulation* Global E-type system evolution is feedback regulated.
IV	*Law of Conservation of Organizational Stability* The work rate of an organization evolving an E-type software system trends to be constant over the operational lifetime of that system or phases of that lifetime.
V	*Law of Conservation of Familiarity* In general, the incremental growth (growth rate trend) of E-type systems is constrained by the need to maintain familiarity.
VI	*Law of Continuing Growth* The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime.
VII	*Law of Declining Quality* Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining.
VIII	*Law of Feedback System* E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.

Table 2.5: *Laws of software evolution, reformulated in 1996 [Leh96] and empirically revalidated in 1997 [LRW+97]. This form remained in the book published in 2006 [MFRP06]. This can be considered the current formulation of the set of the Lehman's laws.*

2.2.3 Law of increasing entropy/complexity

The second law of software evolution says that unless control actions are taken, the complexity of the produced software will increase. This is because of the first law, the continuing change process deteriorate the system, and makes complexity increase.

It is interesting to remark that at its first publication (table 2.2 on page 16), Lehman used the term *entropy* rather than *complexity*. Besides that change, the law has persisted over the years[3]. In its latest form, Lehman added that in addition to showing an increasing complexity, the system *becomes more difficult to evolve*.

It seems that Lehman was wondering about what the meanings of entropy and complexity were. About this issue, Pirzada remarks that "the other significant change has been the replacement of the term *entropy* by *complexity* to reflect the research being conducted into program complexity at the time" [Pir88]. In his 1974 lecture, Lehman explains that the second law considers complexity regarding the structure of the system (it would be today called architecture). Thus, he distinguishes between three levels of complexity: internal, intrinsic and external. Internal is about the structural attributes of the code. Intrinsic is about the dependences between different parts of the program[4]. Finally, external complexity is a measure on how easy is to understand the code provided that there is documentation available. Belady reviewed the main papers on complexity at the time and provides an overview of the research that was conducted on the topic during the seventies [Bel79] (reprinted as [Bel85]). In any case, the interplay between complexity and entropy has been object of intense research [Har92, Dvo94, KSW95, Roc96].

The main idea behind this law is that with frequent changes come disorder, and with disorder the system is more difficult to comprehend and hence to change. In Lehman's terms, the changes degrade the structure of the system, making the system more difficult to maintain. This increasing difficulty to change means that the system becomes more complex, although it does not necessarily mean that syntactic complexity increases. For instance, the additional difficulty may be caused by the degradation of the conceptual consistency of a design [Dvo94].

It has been proposed to use the *semantic entropy* as another dimension of the complexity of software [EGWEH02]. Traditionally, complexity has been measured using syntactic metrics (for instance, the well known McCabe's cyclomatic complexity [McC76]). In the opinion of Etzkorn *et al.* [EGWEH02], complexity has two aspects: code complexity and implementation domain complexity. Classical metrics measure only the code complexity. The implementation domain complexity is related to the semantic content of the code. For instance, a code with obscure variable and function names will be more difficult to understand, and therefore more complex. The mentioned work argues that

[3]Again, in its latest form, the concept of *E*-Type software is used, and the text has changed too, but the meaning of the law is the same.

[4]A large program has a defined task, that is divided in many subtasks. The intrinsic complexity is related to the relationships and the extent of those subtasks

"a complexity metric that provides a code complexity analysis [...] by the use of a syntactical analysis alone will never provide a complete view of complexity."

2.2.4 The third law and the notion of feedback

This law has kept changing over the years (see tables 2.2, 2.3, 2.4 and 2.5), although the rationale behind its conception has remained the same. In its original publication, it states that the growth of software systems is statistically smooth. In other words, that statistics may be used in order to model and represent the software process for purposes of planning, forecasting and improving.

When working at IBM, Lehman gained access to a subset of metrics regarding the evolution of OS/360: the size of the system (in number of modules), the number of modules added, removed and changed, the releases dates, the amount of manpower and machine time used and costs involved in each release.

When he plotted these variables over time, the plots resulted to be apparently stochastic. However, when averaged, the variables could be classified in two groups: some of them grew with a smooth profile, and some others showed some kind of conservation (either remained constant, or with a repeating sequence). These findings suggested that the dynamics of software evolution was feedback driven.

In his lecture in 1974 [Leh74], where Lehman describes the data to which he gained access in IBM), he already suggested to use regression techniques, autocorrelation plots and time series analysis for the study of software evolution. Although the term *feedback* was not introduced in this law until 1996 [Leh96], the idea of software evolution being a feedback driven process already appeared in its original conception. Actually, this law occupied most of the discussion in the 1974 lecture.

In its current formulation, the notion of feedback is explicitly mentioned, and the text is much simpler, just remarking that evolution is feedback driven. In the latest form of the laws, the feedback dynamics was converted into a law itself, and the third law only mentions that evolution is self-regulated. This self-regulation is made through a feedback process.

But feedback is not the only effect behind the empirical findings. Software evolution is statistically smooth because its trend does not greatly vary in the history of the system. Besides feedback, Lehman introduced two more effects in his paper in 1978 [Leh78] (reprinted as [Leh85c]): inertial and momentum effects. The net result of this three effects is that "for maximum cost-effectiveness, management consideration and judgement should include the entire history of the project with the current state having the strongest, but not exclusive, influence" [Leh78] (extracted from the reprinted edition [Leh85c]). In other words, as early as 1978, Lehman found out that software evolution is a short memory process, although some longer term effects may have an influence in the evolution (for instance, periodical events with large periods).

2.2.5 The fourth and fifth laws

The laws reviewed so far can be considered general. The fourth and fifth laws can be understood as boundary conditions for the third law, corresponding to the software development teams and environments usual at the time.

The fourth law states that the work rate remains invariant over the lifetime of the project. In other words, and using the terms of the third law, there is a statistically invariant trend in the activity rate of the project. That invariance is probably due to the feedback process governing the evolution of the system. The fourth law is a boundary condition of the third law, corresponding to the environment where Lehman obtained the empirical evidences that led to the formulation of the laws.

Regarding the fifth law, the case is more interesting. In its original publication(see table 2.3 on page 17), the law states that there is a *safe* growth rate, and that the interval between releases should be kept as long as possible (so the growth rate is kept under control). In other words, this law is a *release when ready* principle.

This law has later changed the *release when ready* principle by the invariance of the content of the releases (table 2.4 on page 18), or in its latest form by the constraints in the growth rate of a project (table 2.5 on page 19).

2.2.6 Law of continuing growth

This law appeared in the latest form of the laws of software evolution (table 2.5 on page 19). It establishes that among all the types of changes that will take place (as predicted by the first law), one corresponds to increasing functionality.

Originally Lehman attributed this need for an increasing functionality to missing requirements in the original specification of the system. Because of time, budget and other constraints, not all the requirements can be included in the initial specification. After some time, users ask for those discarded requirements to be implemented, which causes the continuing growth.

2.2.7 Law of declining quality

This law was also introduced in 1996 (table 2.5 on page 19). Here quality is referred to user satisfaction. Unless work is done to avoid it, as time passes, users satisfaction will decline.

The first law establishes that software will suffer continuous changes during its lifetime. Those changes degrade the architecture of the program, increasing the difficulty to make new changes (second law). At the same time, users will request new features, and the functionality of the system will grow (sixth law). This will happen in an environment of increasing complexity and constrained growth (fifth law). Users request new features because their original requirements were not originally included in the system.

This scenario clearly leads to a decreasing user satisfaction (and hence quality), unless a strategy is taken to avoid it. Therefore, we can consider this law a corollary of the others.

2.2.8 Law of feedback system

Lehman found that evolution was driven by a feedback dynamics in his early studies during the seventies. It was already mentioned for instance in the original publication of the laws [Leh74]. However, until 1996 he did not decide to include this notion as a law itself [Leh96]. The text of the law is very simple, stating that the software evolution is a multi-level, multi-loop and multi-agent feedback system.

Lehman derived this law observing the organizations that built the software that he used as case studies for his empirical work. For instance, it was usual that many managerial levels were implied in the project. This, and the apparent conservation of some quantities (already mentioned in subsection 2.2.4) led to the stating of this law.

2.2.9 Principle of software uncertainty

Summarizing, the field of software evolution has kept evolving since its birth in the late sixties and early seventies (see figure 2.1). The first publication contained three laws, about the changing nature of software, its increasing entropy because of change, and the smooth evolution trend that makes it possible to use statistics to study evolution.

The summary of the global idea behind the theoretical framework that Lehman has shaped over the years is condensed on the *software uncertainty* principle, that states, in Lehman's own words [Asp93]:

> No *E*-type program can ever be relied upon to be correct.

In other words, software is never finished, and keeps evolving in order to fix defects introduced in the previous programming activities, and to take into account new demands of users. As software is a model of the world, and the world changes, the only fate of software is to change or to die.

2.3 Empirical studies of software evolution

In the previous chapter, I have reviewed the theoretical concepts of software evolution, and I have mentioned some works that faced those theoretical predictions against the facts provided by empirical studies.

In this section, I discuss and summarize some of the attempts to empirically validate the laws of software evolution. The main approach has been statistics-based, although the actual statistical tests vary in the different works. I also include a review of how these empirical studies have changed because of the availability of larger datasets.

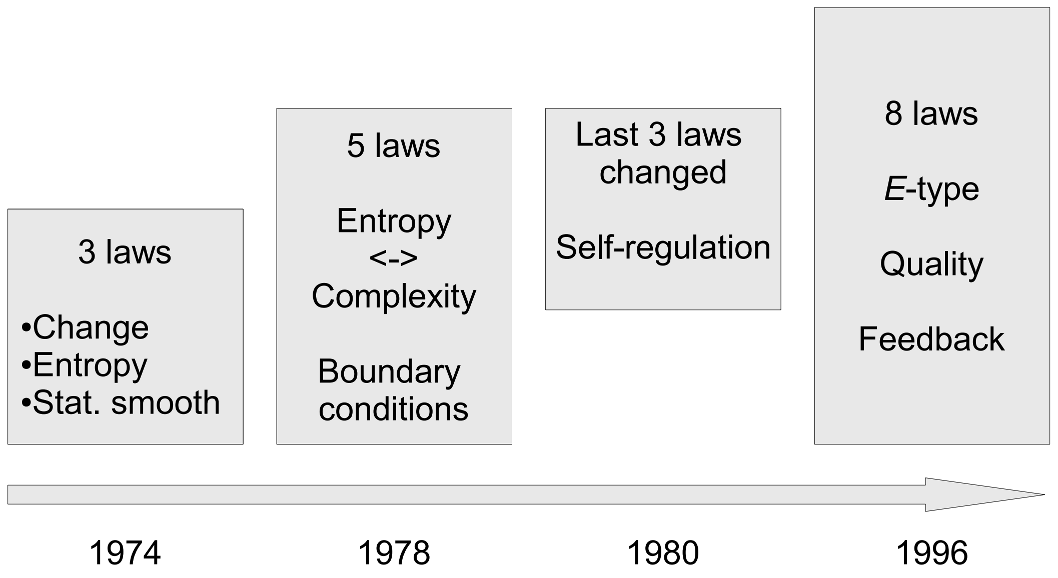

Figure 2.1: *Diagram of the evolution of the laws of software evolution. The laws have increased from three to eight laws. In their latest form, the notion of feedback is the key concept. The term* E*-type has been introduced in the text of the laws, to remark that the laws are only applicable to that kind of software.*

2.3.1 Laws of software evolution

The first work to address the issue of the validity of the Lehman's laws of software evolution using a statistical approach was the PhD thesis by Pirzada [Pir88]. In that thesis, he studies the evolution of some commercial flavors of UNIX. At the time of the study, there were some concerns about the evolvability of those versions of UNIX. Pirzada applied the methods proposed by Lehman *et al.* [LB85]. His dissertation was not only a study on the evolution problems of UNIX, but also a critical evaluation of the validity of the predictions of the laws of software evolution.

The versions of UNIX under study were divided in three branches (or *streams*, in the terminology used by Pirzada): research, academic, and supported and commercial. The versions of Unix that were part of each *stream* are the following:

- **Research stream**
  The original Unix version created by Ken Thompson and Dennis Ritchie in Bell Labs. Pirzada studied 9 versions of this flavor of Unix, released between 1971 and 1987.

- **Academic stream**
  In this stream, Pirzada included the version of Unix developed in the University of California at Berkeley, by the Computer Systems Research Group (CSRG). The last release considered by Pirzada, 4.3 BSD, appeared in 1986.

- **Supported and commercial stream**
  In this stream, Pirzada includes the versions of UNIX developed by the Unix Support Group (USG), a small team created by the Switching Control Center System, an internal department of Bell Systems.

The conclusions of Pirzada were very clear: only the supported and commercial stream was evolving according to the laws of software evolution (mainly, slowing down because of increasing complexity). More interestingly, the academic and research streams were growing rapidly, accelerating in some cases (BSD). Their complexity was not growing through the different releases. The conclusion by Pirzada was that processes in strongly commercial environments are more constrained, and much more likely to exhibit structural deterioration. In other words, software developed in intensive commercial environments is more likely to evolve conforming to the original formulation of the laws of software evolution.

One of the main conclusions of Pirzada is that only software developed in commercial environments fulfills the third, fourth and fifth laws (as formulated at the time, see table 2.4 on page 18).

Regarding the third law (see section 2.2.4 on page 21), which states that software is a feedback driven process, statistically smooth and predictable, Pirzada defends that is not valid for the research and academic streams, and so that only software developed in commercial environments can be studied using a statistical approach. He even suggested a new text for the law taking in account this. The conclusions of this thesis do not agree with the new law proposed by Pirzada: I will show how Time Series Analysis can be used to forecast the evolution of software projects because their evolution dynamics is a short memory process (at least, bounded to the case studies considered in this thesis).

Regarding the second law (see section 2.2.3 on page 20, and table 2.3 for the formulation at that time), again Pirzada finds evidences conforming to this law only for the commercial stream. But what is even more interesting is that the increase in complexity is verified only after the commercialization of the products. The conclusion of his thesis is that the commercial pressures enforce constraints to the growth of software and foster deterioration of the system. Pirzada dedicates a whole chapter in the thesis (entitled *Under pressure*) to explain why this behavior is verified only in products that experiment a commercialization process.

However, this work was not the first one to highlight some controversy about the universality of the laws of software evolution. Pirzada reports about some works that had previously validated the laws of software evolution using an empirical and statistical approach. From all the works cited by Pirzada ([Cho80, BT79, Kit82, Law82]), I could access only to a paper by Lawrence [Law82]. That work is a statistical validation of the Lehman laws using the case studies reported by Pirzada (see table 2.1 on page 15). Those case studies had been used in previous empirical studies [Cho80, BT79, Kit82], but Lawrence used different statistical tests, concluding that the third, fourth and fifth laws were not fulfilled. First and second laws were valid though.

2.3.2 FEAST and the nineties view

The software evolution field continued to be focused on the issue of the validity of Lehman's laws during the nineties. At that time, Lehman leaded the *Feedback, Evolution and Software Technology* (FEAST) research project (1996-1998, extended 1999-2001), funded by the Engineering and Physical Sciences Research Council of the United Kingdom. The main goal of the project was to show that software evolution was a multi-loop feedback process. As a side effect, the project intended to remark that software evolution was a subject of research in its own right.

The results of that project were summarized in a paper by Lehman and Fernández-Ramil [LR02]. The studies done under the scope of the FEAST project provided support for the laws of software evolution, either empirically or indirectly. However, some conflicting cases reported in the literature were mentioned: the case of the OS 360/370 defense system (reported in the same publication), and the work by Godfrey and Tu on the growth of Linux [GT00].

The report on FEAST also recognized that for the first time during the study of software evolution, the systematization and formalization of the field appeared possible.

There were many other independent studies that verified the laws of software evolution. For instance, the study by Anton and Potts [AP01, AP03], that establishes that software evolution is a case of *punctuated equilibrium*: instead of a gradual change, evolution occurs in discrete bursts, where bunches of features are introduced at the same time. According to [LR02], that mechanism supports the laws of software evolution. More recently, a similar mechanism has been proposed by Wu [Wu06, WHH07] (we will review these works later in this chapter).

Other work supporting the conclusions of Lehman and coworkers is [Aoy02], that suggests that evolution should be studied in stages. This is, the life cycle of a software project goes through different stages, and the evolutionary behavior of a project is different in each stage [RB00]. This view goes beyond the traditional distinction between the development and the maintenance or evolution stages: the software life cycle contains five stages (development, evolution, servicing, phase out and close down). In other words, the evolution stage is composed of four stages. In each one of those stages different activities are performed and different tools are used. This is reflected in a different evolutionary behavior for each stage. This model of *staged software evolution* has been adapted for the case of libre software [CGBHR07]. Indeed, the evolutionary behavior of libre software has been empirically proved to be different in each stage of its life cycle [CM07].

The nineties supposed a renewed interest in the study of software evolution. The classical work by Lehman was revisited, and many studies backed up the predictions of the laws of software evolution. However, the field of software evolution evolved itself, in order to face the shift in the programming practices.

In 1994, the invited keynote of the International Conference on Software Maintenance was titled *Dimensions of Software Evolution* [Per94] (later revised and republished as [Per06]). The main conclusion of that paper was that evolution depended not only on the age, size or stage of the project, but also on the nature of the environment. This is interesting, because all the research on the validity of the laws have been related to the nature of the environment where each study took place. For instance, the case of libre software is directly related to the nature of the environment where software is developed.

These works led to the reformulation of the laws of software evolution in 1996 [Leh96] (the current formulation of the laws remains the same since then). This last formulation of the laws was revalidated by Lehman *et al.* [LRW$^+$97], finding empirical support for all the laws but the fifth (conservation of familiarity, a boundary condition for the law of feedback) and the seventh (declining quality). Other studies at the time found empirical support for the laws of software evolution too [GJKT97], although by that time there was a lack of empirical studies in the field of software evolution [KS99].

However, soon some new works started to remark that some of the Lehman's laws of software evolution seemed not to be valid for the case of libre software [GT00, GT01]. Those works started a prolific line of research, about the differences between software development in open and distributed communities and within industrial and closed environments.

Summarizing, for software evolution, the nineties decade saw:

- The empirical validation of Lehman's laws, and their reformulation, to keep them synchronized with the new software development tools and practices [Leh96, LRW$^+$97].
- The suggestion of using segmentation to study each one of the stages of the evolution of software projects [RB00].
- The suggestion of new mechanisms (for instance, punctuated equilibrium) that could serve as a base for a theory of software evolution [AP01].
- At the end of the nineties, the re-arising of the validity of the laws as a topic of research, that was first addressed in the eighties [Law82, Pir88] but had not previously attracted attention during this decade [GT00].
- The highlight of the lack of empirical studies on software evolution [KS99].

2.3.3 Libre software and software evolution

The paper by Godfrey and Tu started a new line of research, about the evolution of libre software projects [GT00]. They studied the case of the Linux kernel, and found that Linux was growing according to a quadratic curve. In a later work, they found that some time later Linux was still growing with that fast pattern [GT01].

The methodology used by Godfrey and Tu was based on the measurements of the lines of code of all the releases of Linux. They studied the evolution both at the global level and at the subsystem level, as it had been suggested by Gall *et al.* [GJKT97]. Among other evidences, they found that cloning could be a possible explanation of the fast growth.

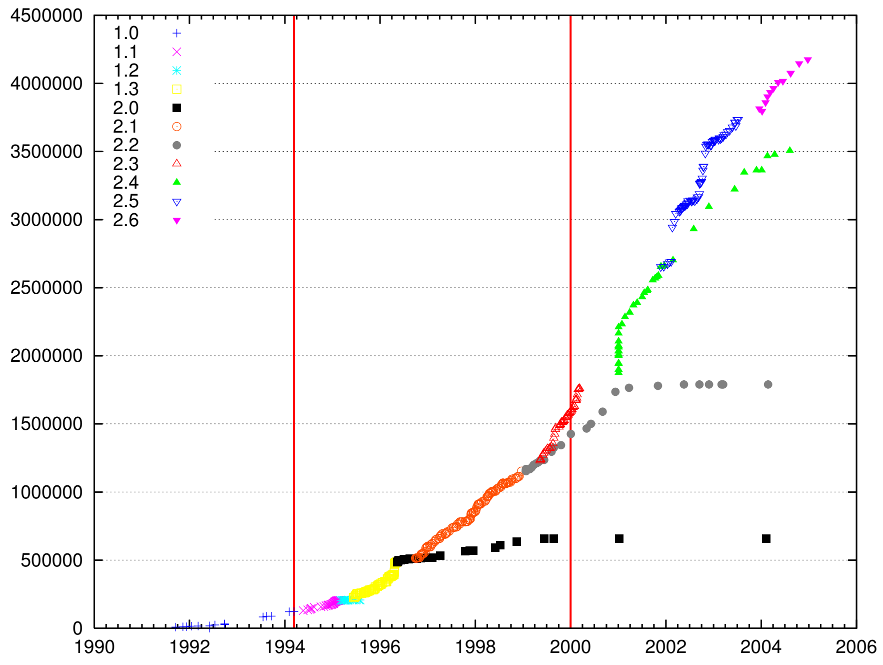

Figure 2.2: *Growth in SLOC (lines of code removing blank and comment lines) of the Linux kernel. Each color is one of the branches of Linux. The overall growth follows a quadratic equation. Reproduced from [RAGBH05].*

Five years later, Linux was still growing with a quadratic pattern [RAGBH05]. However, this time the coefficient of the quadratic term had increased. In that study, Robles *et al*. repeated the subsystem studied performed by Godfrey and Tu [GT00]. Using that methodology, they found that most of the subsystems were growing with a linear profile, and that the quadratic profile of the overall growth was the result of the addition of many linear segments.

In other words, rather than a quadratic curve, the growth profile was a polyline formed by all the linear segments of each one of the subsystems. Figure 2.2 shows the overall growth curve for Linux, as of 2005. The curve was found to be quadratic. However, as said above, the quadratic shape was the result of adding many linear segments, with different starting points. Each linear segment corresponded to a subsystem of Linux, or to a stage in the evolution of a subsystem (see figure 2.3 on the facing page). When going deeper in the tree structure of Linux, each subsystem had a linear growth, or at least, were formed by linear segments.

This quadratic curve (or more precisely, polyline of the subsystems' segments) was one of the topics of one of the first massive studies on the evolution of libre software [Koc05]. In that paper, 8,621 projects from SourceForge.net were studied. For them, the quadratic profile was found to model the data better than the linear one.

The same topic was addressed in [HRGB$^+$06]. In that paper, the case studies were 13 large projects obtained from the largest packages of the Debian GNU/Linux distri-

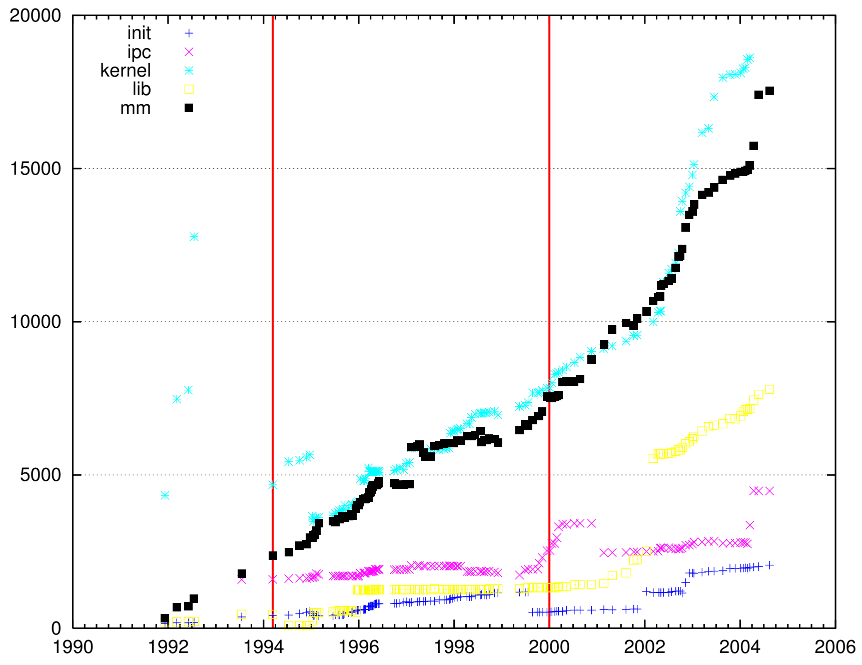

Figure 2.3: *Growth of smaller, core subsystems of Linux. Some of these subsystems are growing linearly. In general, when looking at the subsystems level, the growth of each subsystem is linear. The sum of many linear segments with different starting points gives the overall quadratic shape. Reproduced from [RAGBH05].*

bution. The predominant profile was quadratic, although some of the cases were linear and some other sublinear (verifying this case the behavior expected according to the laws of software evolution).

These findings suppose that libre software is growing faster than predicted by the laws of software evolution, and hence that those laws are not fulfilled in the case of libre software. However, these affirmations have not always been found to be validated [PSE04].

Fernández-Ramil *et al.* [FRLWC08] summarize the main works on the topics mentioned above, and on the empirical studies of the validity of the Lehman's laws for the case of libre software. Based on those works, an analysis of the validation or invalidation of the laws done so far for the case of libre software is provided. Summarizing, not all the laws find empirical support, but none of them has been empirically invalidated to date.

However, the most recent empirical studies on evolution and libre software have shifted from the issue of the validity of the laws of software evolution, to the topic of modelling evolution. For instance, Capiluppi *et al.* wrote a series of papers in 2004 [CMR04b, CR04, CMR04a] where we can observe this shifting in the topics. In the first of the cited papers [CMR04b], Capiluppi *et al.* study how a libre software project evolves, measuring its size using different units. As well as metrics traditionally used in these studies, they add the depth and width of the tree of source code of the project. The conclusions are that the project is growing with a linear trend. However, the depth of the tree remains constant over the time, and it only grows in width. Although not explicitly mentioned, the conclusions match the results expected using the laws of software evolution. In this paper, there is no special emphasis in trying to obtain a model for the evolution.

The next paper in the series [CR04] is a study of the evolution of size, complexity (using the McCabe and Halstead's complexity metrics) and activity rates at different levels of granularity (directories, files and functions). The conclusions of that work are the same than in the previously mentioned paper, but this time for both case studies. The results were consistent with Lehman laws.

But the most interesting paper in that series is the third one [CMR04a]. In that paper a set of 25 libre software projects was studied, using a methodology similar to the two previously mentioned papers. The goal was different, though. The previous papers were in the line of testing whether or not the case studies were evolving according to some hypothesis, formulated taking account Lehman's predictions. This time, the aim was to try to predict and model the evolution of those projects, based on the commonalities and patterns found in the 25 projects. In other words, that paper clearly shows the shift in the topics of the empirical studies of the evolution if libre software, towards trying to obtain patterns, commonalities and models of the evolution.

2.3.4 The importance of metrics for validation studies

In the first empirical studies on software evolution made by Lehman, Belady and others, they had no direct access to the source code and the change records. They gained access only to some datasets provided by the companies proprietary of the systems, that contained only a few metrics. For instance, in the original work by Lehman in 1969 (republished as [Leh85a]), he had only the following data: size of the system in number of modules, number of modules added, removed and changed in each release, releases dates, information on manpower and machine time used and costs involved in each release. Having only an unique point of data for each release, he decided to use the release number as pseudo-unit of time, that he called *Release Sequence Number* (RSN). He decided to use number of modules as the base unit for size as well.

These units have persisted in later revisions of the work by Lehman and others [Leh96, LRW^{+}97, RL00, LRS01]. Empirical studies about the evolution of libre software have used Source Lines of Code (SLOC)[5] as size metric [GT00, RAGBH05]. These works started the controversy about the validity of the Lehman's laws for the case of libre software. However, the metrics (both for time and size) were different. Godfrey uses SLOC as size metric, Lehman number of modules. The data used by Godfrey was obtained over natural time, and the data used by Lehman over RSN.

[5]Basically, SLOC is number of text lines of a source code file, removing blank and comment lines. Section 3.2.3 on page 48 reviews that and other size and complexity metrics.

Lehman qualified the study by Godfrey [GT00] as an anomaly [LRS01], and argued that the studies were not comparable because of the metrics.

In a previous work [LPR98], Lehman had talked about the suitability of lines of code as a size metric for evolution studies:

> A lower level metric, lines of code (locs, or equivalently klocs) much beloved by industry, does not have semantic integrity in the context of system functionality. Moreover, even where programming style directives are laid down in an organisation, the numbers of locs produced to implement a given function are very much a matter of individual programmer taste and habit.

We addressed this problem on metrics in an empirical study done before the studies shown in this thesis [HRGB^{+}06]. In 13 large libre software projects, the evolutionary profiles and growth curves were the same regardless the metrics used (number of files or SLOCs). There were high correlations among number of files (or modules, in the terminology used by Lehman) and number of SLOCs. In this thesis, we have extended that study to a sample of 6, 556 projects, and added some complexity metrics. The same conclusions hold, as we will show in the next chapters (there is a preliminary version of the results already published [HGBR07b]).

Regarding time units, although there has not been too much specific research on the topic, some studies recommend to use natural time rather than RSN when natural time data are available [BKS07].

2.3.5 Towards large scale investigations

If we observe the empirical works presented in this chapter, there is a certain trend to increase the number of case studies used in empirical studies of software evolution. As I already showed in chapter 1, the increasing availability of software repositories, thanks to the growth of the libre software community, has influenced the size of the empirical studies on software evolution.

If the first of the studies (in 1969 by Lehman, republished as [Leh85a]) considered only one case study, this thesis (2008) uses three samples of 12, 010, 6, 556 and 3, 821 case studies (for different kinds of analysis).

Between those two points, there have been many empirical studies that varied in the number of considered case studies. In 2007, Kagdi *et al.* [KCM07] made a thorough survey of the different approaches used to mine software repositories for software evolution studies. Based on that work, and on the works reviewed in this chapter, we have represented the evolution of the number of case studies in empirical works on software evolution (see figure 2.4). The graph shows only some selected case studies, that we consider representative of the size of the empirical studies performed by that date. The vertical axis shows the number of case studies in logarithmic scale. One of the curves includes only selected papers on software evolution. The other one includes two relevant

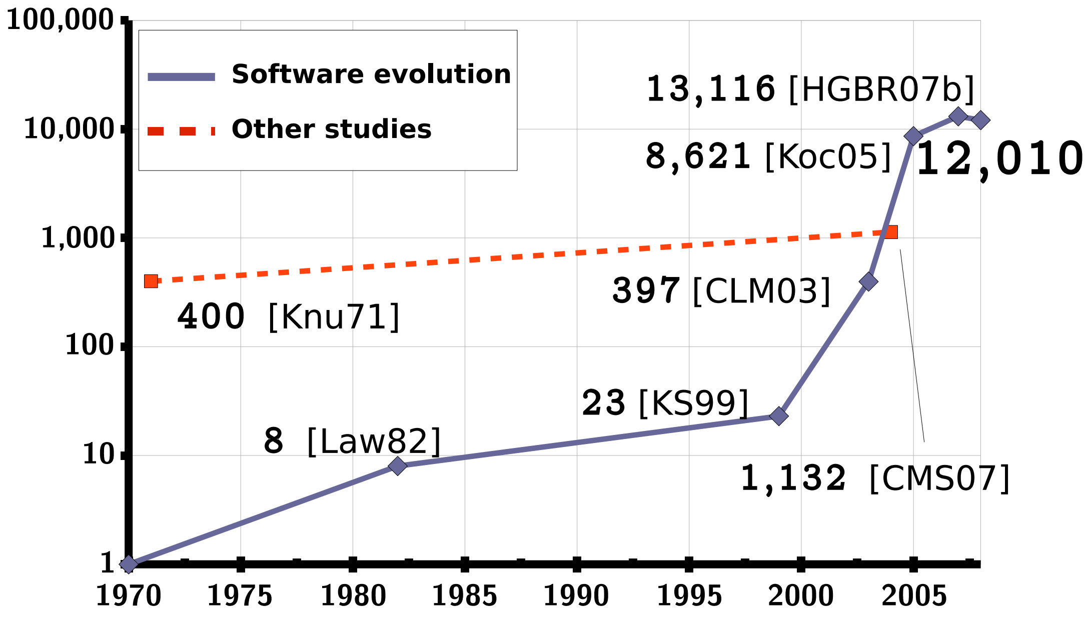

Figure 2.4: *Number of case studies in empirical studies on software evolution, over the years since the original study by Lehman in 1969. Vertical axis shows the number of case studies, in logarithmic scale. Horizontal axis shows the year of the study. This graph shows only some selected papers, based on the works reviewed in this chapter and in those included in the survey [KCM07]. This thesis (2008) analyzes 12,010 case studies, that were obtained from a previous study [HGBR07b] that analyzed 13,116 case studies .*

empirical studies that were made at a large scale, although the topic is not software evolution. The first of those works is a paper written by Knuth in 1971 [Knu71], that studied 400 FORTRAN programs with the goal of improving the design of FORTRAN compilers. The other study [CMS07] studies 1, 132 bytecode programs (originally written in Java), with the goal of helping in the development of future programming languages and in the implementation of compilers. We only mention those works because they are relevant as empirical studies done at a large scale, but we insist in that they are not software evolution studies.

The first point in the software evolution curve corresponds to the original study by Lehman in 1969. The last point corresponds to this thesis. The first cited work [Law82] is a validation of the laws of software evolution using a statistical approach. The paper analyzed 8 case studies, that were industrial projects coming from different domains. Pirzada reviewed that work in his PhD thesis [Pir88], and extended some of the information about the case studies (see table 2.1 on page 15).

The second work is a highly influential paper for the field of empirical studies of software evolution [KS99]. It includes a review of all the relevant literature in the topic, classifying the works according to the statistical approach used to study the evolution of software. The paper tests some techniques, among them time series analysis. That was done in a set of 23 proprietary systems. One of the main conclusions was the highlight of the lack of empirical studies for software evolution.

By those dates, and during the following years, many papers studied sets of software systems of similar sizes. But in 2003, Capiluppi *et al.* [CLM03] performed one of the first large scale empirical studies, by studying 397 libre software projects, that were obtained from Freshmeat.net (a well known web site that compiles information about libre software projects). The authors gathered different properties of each project, at different moments of time, and analyzed the relationships among the properties, and how the properties changed over time. Although not exactly an empirical study of software evolution, it is a relevant example of massive empirical studies.

The next paper has already been reviewed in this chapter [Koc05]. The analysis was performed in a set of $8,621$ projects, obtained from SourceForge.net (that contains in total over $100,000$ projects).

The next paper showed in the mentioned figure analyzes $13,116$ projects that were obtained using the packages system of FreeBSD, an operating system that includes many packages with source code from third parties [HGBR07b]. The conclusions of the paper were mainly two: size and complexity metrics are highly correlated (tested only for the case of the C language), and the statistical distribution of size and complexity is a double Pareto. That paper corresponds to an early version of the work and conclusions presented in this thesis.

2.3.6 The statistical properties of the size of software

The availability of large datasets about software development makes it possible to use a *survey approach* to find out which are the usual dimensions and structure of software.

Traditionally, the kind of questions that empirical studies try to solve is how we can design and build better software, with less defects, consuming less resources. However, if we do not know which are the usual properties of software, it can be hard to judge the quality of a particular program, because we do not have anything to compare with.

This kind of studies is not new ,though. In 1977, Clark and Green studied the use of list structures in programs written in Lisp [CG77], and found that the pointers to *atoms*[6] followed a Zipf's law (a power law distribution, with a particular value of the values of the parameters of the distribution). The study was static, just looking at source code and not trying to run the programs to test their efficiency.

One of the classical papers in computer science is also an empirical study of this kind: the study of FORTRAN programs by Knuth in 1971 [Knu71]. In that paper, he tried to find "what programmers really do", using a survey of randomly selected FORTRAN programs, with the goal of helping in the design of FORTRAN compilers. The point was that compilers are designed for the most typical case, but the most typical case could not be known without studying actual FORTRAN programs in a large scale.

[6]In Lips, atoms are immutable and unique lists. If two atoms in a program have been written in exactly the same way, they represent the same object and both point to that object.

More recent studies have found power laws and other kind distributions in software. For instance, Baxter *et al.* studied a set of 56 applications written in Java, and calculated 17 metrics for each application [BFN$^+$06]. Some of the metrics were distributed as a power law, while some others were not. However, the results were not consistent through the different case studies. Some of the metrics were distributed like power laws in some case studies, and with other shapes in the rest of applications.

Louridas *et al.* found power laws in the network of dependencies between software packages (Java programs, FreeBSD ports, etc) [LSV08]. In their conclusions, they suggest that power laws might not be the only distribution that could be fitted to the empirical data. They also give some practical advice for software reuse, quality assurance, library design and runtime module loading optimization, based on their empirical findings.

Power laws seem to have gained attraction in the last years. As remarked by Fabrikant *et al.* [FKP02], it has been maybe too much attraction:

> Power laws [...] have been termed "the signature of human activity" [...]. They are certainly the product of one particular kind of human activity: looking for power laws.

Mitzenmacher [Mit05] proposes a research agenda for studying power laws. This kind of empirical studies can be classified in five types of investigations: observe, interpret, model, validate, and control. So far, there have been papers on the first three topics (observe, interpret and model), but not on the rest of them. In Mitzenmacher's opinion, at this moment "observing a power law in itself no longer seems sufficient for publication in networks."

In any case, for the particular case of software, there are still very few works observing power laws (or any other type of distribution). This kind of distributions has deep implications for empirical research. For example, as highlighted by Baxter *et al.* [BFN$^+$06], power law distributions have infinite mean and variance. Therefore, when studying a sample of software projects, its mean and variance cannot be used as an estimator of the mean and variance of the population.

Power laws are not the only distributions found in software. When studying empirical data, it is easy to misidentify it as a power law, when it is actually a lognormal distribution, or any other type of distribution with a large tail [Dow05]. In the case of software, both power law and lognormal distributions have been found [CMPS07, AH06]. Concas *et al.* [CMPS07] argue that the presence of those distributions of software denotes that the programming activity cannot be simply modeled as a random addition of independent increments but exhibits strong organic dependencies on what has been already developed.

The misidentification of power law and lognormal distributions is the topic of paper by Mitzenmacher [Mit04a]. He explains that there is a debate about whether filesystems are better explained using power law or lognormal distributions. Because of that debate, he investigated on the issue and discovered that it has repeated many times in the past in other fields (economy, biology, etc). In order to provide arguments for the file-systems case, he presents the properties of each kind of distribution, and the generative processes for both distributions.

In this respect, there has been an intense research on the statistical distribution of file-systems size. In 2001, Downey [Dow01] presented empirical data that showed that the distribution was lognormal, contrarily to previous research that had been shown that it was a power law distribution. He also presented a model that explained why these distributions appear in file-systems. This model shares some similarities with the processes that occur in a source code tree while the system is evolving. It has some limitations though. For instance, it assumes that the tree starts with a single file, new files are always created by copying or modifying existing files and files cannot be deleted.

However, in 2004 Mitzenmacher [Mit04b] presented a new analysis for the size of file-systems, and obtained that they are better modelled using a *double Pareto distribution*. This kind of distribution has a Pareto tail, but a body close to lognormal. The Pareto tails may be present at the extremes values (both low and high). Based on the appareance of that distribution, Mitzenmacher proposes a generalization of the Downey's model, that he calls the *Recursive Forest File Model* (RFFM). This model overcomes the limitations of the Downey's model. For instance, it allows the addition and deletion of files, and the starting number of files and their structure are arbitrary. The RFFM also shares many similarities with the evolution of a source code tree, and with the typical operation of a version control system. Because it has not the limitations of the Downey's model, the RFFM could be used to simulate the behavior of version control systems in real projects. We discuss about the suitability of this model for software evolution simulations in section 5.2.1.

All the works presented in this section use a statistical approach, with the goal of obtaining the shape of the distributions of typical parameters like size. They are the link between the empirical works that try to validate the laws of software evolution, and the works that try to obtain a model for software evolution. In the next section, among other studies, I review some of the works that have based on the statistical properties of software to obtain a model of software evolution.

2.4 Modelling evolution

In recent years, empirical studies on software evolution have shifted from trying to validate Lehman's laws, to the attempt of obtaining a model for software evolution. There are three main approaches when trying to obtain a model for software evolution: *statistical*, *physical* and *blended*. The statistical approach cannot provide explanatory models, this is, those models can tell us how a software project will evolve in the future, but not why. The physical approach aims to obtain explanatory models. However, to our knowledge, although there are some proposals of physical models, none of them have been tested to be accurate enough in a wide range of cases. Finally, the blended approach tries to obtain models for evolution based on the statistical properties of software. I describe in this section two models of this kind: the Sand Pile Model, and the Maintenance Guidance Model.

2.4.1 Physical models

The first model for software evolution was proposed by Belady and Lehman in 1971 [BL71] (republished as [BL85]). That paper presents a macro model for evolution, taking into account the empirical findings of Lehman in 1969 [Leh85a]. The model reproduced the same kind of curves than the empirical studies, and was obtained based on theoretical considerations on the programming process. All the results were obtained at an aggregated level. Some hints were provided to be considered for a hypothetical micro model.

In 1974, Lehman included an overview of some macro models similar to that mentioned above [Leh74] (republished as [Leh85b]). All the models included in the lecture were obtained using the *physical* approach. However, in the same lecture Lehman suggested to use statistics and time series analysis to analyze the empirical results that he obtained in 1969.

Several papers addressed this topic during the seventies (for instance [BL76]). One of those papers [Woo80] was included in the book edited in 1985 [LB85]. In that paper, Woodside presents a model that was much more thorough than the previous ones. It is also a macro model. In essence, it tries to balance between *progressive* work and *anti-regressive* work. Progressive work corresponds to tasks done towards increasing the functionality of the system. Anti-regressive work corresponds to tasks done to keep complexity under control. Based on the balance equations, the model provides values for parameters at the global level (for instance, number of modules).

Other macro models similar to the first ones are those proposed by Turski [Tur96, Tur02]. The first model was described using difference equations, and the parameters of the model were discrete values [Tur96]. Later, the model was generalized to continuous variables and differential equations [Tur02]. The model by Turski is based on two assumptions, which derive from the Lehman's laws:

- The growth of the system is inversely proportional to the system's complexity
- The complexity of the system is proportional to the square of the size of the system

Other *physical* models have tried to mimic natural processes. For instance, Robles *et al.* presents a model based on the *stigmergy* concept [RMGB05]. It assumes that communication between individuals happens trough stimuli caused by changes in the environment, and not by directly exchanging information. The model has been used to find out how developers join projects, and how the work by those developers affects the overall evolution of the project.

Other model similar to the first attempts made by Lehman and Belady [BL71] is the model proposed by Antoniades *et al.* [ASAB02, ASS$^+$04]. It tries to describe all the processes found in a typical libre software project using differential equations. The output of the model is the future values of some parameters of the project, at different levels of granularity. A similar model by Dalle and David tries to simulate how resources are allocated in open source software projects [DD03], using a reward mechanism that is driven by reputation of software developers within the community.

2.4.2 Statistical models

The two assumptions made by Turski for his models [Tur96, Tur02] imply that the growth rate must decrease over time. First Godfrey and Tu [GT00], and later Robles *et al.* [RAGBH05] found case studies where the growth rate was increasing instead of decreasing. The authors of those works argued that those case studies did not fulfill some of the Lehman's laws. In other words, the models proposed by Turski could not be used for those case studies.

The authors that found those cases applied linear regression to obtain simple models of the evolution of those cases. Linear regression is the most common technique in the literature to obtain statistical models of the evolution of software [GT00, RAGBH05, Koc05, FNPAQ00].

Other statistical technique used to model evolution is principal component analysis. For instance, Peng *et al.* [PLM07] obtain models for the evolution of some operating systems using that technique.

But in our opinion the most suitable statistical technique to study the evolution of software is time series analysis. This idea is not new. Lehman was the first to suggest it [Leh74]. In the eighties, Chong Hok Yuen suggested the same approach [Cho85, Cho87, Cho88]. For instance, in [Cho88], Chong Hok Yuen used an ARIMA (a kind of model for time series) model to forecast the maintenance stage of a software project, using a sampling period of one month.

However, in a influential paper in 1999, Kemerer and Slaughter showed that using ARIMA models to predict the monthly number of changes did not provide accurate results [KS99]. They argued that the parameter under study was very noisy and close to a random variable, and therefore the model did not provide good results.

ARIMA models have been used in some other papers [ACPM01, CCPV01] with better results. In the case of [CCPV01], the authors applied time series analysis to predict the evolution of the Linux kernel. They took a small subset of all the releases of Linux, and tried to predict the size of future releases. The results were mostly satisfactory, but for some releases the model gave results with large relative errors. In [ACPM01], the authors tracked down the evolution of some clones found in the sources of Linux, and used time series analysis to model the evolution of each clone, again with mostly satisfactory results.

The suitability of time series analysis for the study of software evolution was the topic of a paper by Fuentetaja and Bagert [FB02]. Although it did not provide any particular model, it argued that the statistical tools can be used to obtain valuable information regarding the evolution of a software project.

More recently, Dalle *et al.* [DDdB06] have applied signal processing techniques to gather information from the time series obtained from versioning systems of libre software projects. They were inspired by a previous work [HWH05], which applied time series analysis concepts to visualize historical information from software projects.

2.4.3 Self-Organized Criticality and the sand pile model

The previous sections have differentiated among physical and statistical models. However, there have been some proposals that could be labeled as physical (following our criterion mentioned above), but that we have been formulated based on statistical considerations. I label this category as *blended models*.

In this and the sections, I present a model that explains software evolution as a Self-Organized Criticality (SOC) dynamics, and the Maintenance Guidance Model, based on the fact that the statistical distribution of changes is asymmetrical.

In his PhD thesis, Wu shows that software evolution follows a punctuated equilibrium [Wu06], and that the dynamics of evolution is Self-Organized Criticality [WHH07]. This means that evolution happens in bursts, and it is not a gradual process.

The empirical findings that support these affirmations are the following:

- Power law distributions for the size of the system (in number of developers, and number of commits to the version control system).
- Long range correlations in time series of changes.

If both points are fulfilled, the dynamics is said to be Self-Organized Criticality. There are physical models that explain that kind of dynamics in some natural phenomenons. For instance, one of the phenomenons that follows a SOC dynamics is a running sand pile, whose model was initially proposed by Bak, Tang and Wiesenfedld [BTW88], and that has been adapted to the case of software evolution by Wu.

If a sand pile is formed from scratch, dropping grains of sand over a flat surface, one at a time, the sandpile grows as a set of smaller piles, and the slope increases gradually. However, at a certain moment, and in certain places of the pile, the slope will reach a critical value, and if more sand is added, the pile will slide. If this behavior is repeated over time, the slope of the sand will reach those critical values several times and in several places. This dynamic behavior is call Self-Organized criticality: when the pile reaches a critical state, it reorganizes and its shape changes. The different values of the size of the pile distribute according to a power law, and the time series of the values of the slope shows a long range correlation.

The model has four parameters that may be used to explain the dynamics of the sand pile: driving force, response, system state and relaxing force. Table 2.6 shows the analogy of these parameters made by Wu for the case of software evolution.

Like in the case of a running sand pile, a software system is continuously changing, under the influence of diverse driving forces. Some changes are reactive: they are responses to the requests made by users, or to the defects discovered in the system. In

Parameter	Sandpile Model	Software system
Driving force	Sand drop	Change request
Response	Sand slide	Change propagation
System state	Gradient profile	Release / iteration plan
Relaxing force	Gravity	Stakeholder demands

Table 2.6: *Analogy between the sand pile model and the evolution of a software system. Reproduced from [Wu06].*

Wu's analogy, a change request is similar to a sand drop in a pile. The change itself (the consequence of the request) is mapped to a slide in the pile.

In the case of the sand pile, gravity acts as a regulating force, that once the critical value has reached in a place, forces the pile to slide and lowers the value of the slope in that part of the pile. Wu identifies the stakeholders demands as this relaxing force in the case of software. For the case of the system state, in the case of the pile, it is a matrix that contains the maximum values of the gradients that can be reached in each part of the pile. Wu identifies this system state with the release plan of a software system.

Using this analogy between parameters, and given that the appropriate values for each parameter are known, the sand pile model [BTW88] may be used to simulate the evolution of a software system.

However, this can be done at the overall level. According to Wu, the propagation of changes may be difficult to predict in this case of dynamics. For instance, we could predict how many changes occur in a system, but not where those changes will occur.

Although this model is backed up with empirical findings in 11 libre software projects, no simulation or validation of the accuracy of the model with real cases have been done yet.

2.4.4 The Maintenance Guidance Model

Another example of a *blended* model is the Maintenance Guidance Model (MGM), proposed by Capiluppi and Fernández-Ramil [CFR07]. It is based on two assumptions:

- The distribution of accumulated changes is asymmetrical. This means that most of the changes are concentrated in specific parts of the code base.
- Developers prioritize the parts of the code base where they make anti-regressive effort taking in account the number of changes in the past and the complexity of that part (function, file, etc).

The first assumption is backed up by many empirical works, some of them by the authors of the model themselves [CFR05]. The second assumption is based on the second law of software evolution, which states that complexity will increase unless effort is dedicated to keep it under control. The kind of activities performed to reduce complexity is denominated *anti-regressive* effort.

Based on these two assumptions, the authors argue that those functions that have suffered more changes in the past are more likely to be changed in the future. Given two functions of the same complexity, if developers reduce the complexity of the function that suffer more changes, the impact in the overall productivity will be higher, because it is assumed that after reducing complexity the function will need less changes. This argument is labeled by the authors as the *change-rate criterion for anti-regressive work*.

The authors use different metrics to measure complexity. In their opinion, complexity has three dimensions: functional, structural and coupling. For each one of those dimensions, the authors use the following metrics:

- **Functional:** Lines of code
- **Structural:** McCabe's cyclomatic complexity
- **Coupling:** Fan-in and Fan-out

The base entity used in the study is functions. Thus, they measure all the metrics at the function level, and determine all the changes that each function suffers in all its history.

After measuring all the metrics (complexity and number of changes), the functions are ordered by complexity, and then by accumulated changes. This sorted list contains the candidates to be changed in the next release. This is because of the previously mentioned *change-rate criterion*.

These results are compared against the actual history of changes of eight libre software projects. The accuracy of the model is overall good. However, this accuracy depends on the complexity metric used in the sorting step. In some case studies, some metrics perform better than others, and the same metric does not produce good results for all the cases.

The authors mention that one limitation to apply this model is the computational time for the coupling analysis, that limits *live* application to a version control repository. In my opinion, another limitation is that the model considers the whole number of past changes for each function. If for some reason, a function becomes stable and do not suffer more changes, it will remain high in the sorted list because all the changes made time ago.

2.5 Summary

To summarize the state of the art on empirical studies and software evolution, we include here a classification of most of the research papers reviewed in this chapter. We focus here only on the papers that are empirical studies of software evolution.

We classify the papers according to the main topic (validation, modelling or other), and to the scale of the study. Table 2.7 on the next page shows the classification. It only includes some selected paper, that we have considered relevant.

The first phenomenon that we can identify by looking at that table is that in recent years, the field has gained more attention. This is probably because of the popularization of libre software. That kind of systems seem to be evolving in different ways. That issue has stimulated empirical research on the evolution of libre software ([GT00, GT01, RAGBH05, Koc05, HRGB$^+$06, Wu06]). This is one of the questions that I address in this thesis. I test whether the classical studies by Lehman can be compared with the modern studies on libre software. If they can be compared, then their conclusions, this is, that Lehman's laws are not valid for the case of libre software, are valid.

Another phenomenon that we can observe is that the scale of the studies has increased over the years. As also shown in figure 2.4 on page 32, the first empirical studies were based on very few cases ([Leh74], [Law82]). Nowadays, the increasing availability of libre software repositories, has made much easier to do empirical studies at a large scale ([CLM03, Koc05, HGBR07b]). In the seventies, when the study of software evolution started, it was quite difficult to access to data about the evolution of software systems in a large scale. There were some studies at a large scale though. For instance, the paper by Knuth in 1971 [Knu71] that studies 400 FORTRAN programs. It is not focused on evolution, but it only studies the properties of the source code. The main goal was to improve the design of compilers.

Because of this last point, in this thesis I have studied a large set of case studies. This makes it possible to ensure statistical significance, and to apply the approaches initially suggested by Lehman [Leh74]. Among the approaches that he suggested, we find time series analysis, that I use to study the dynamics of software evolution, and to determine whether previous findings on the topic [Wu06, WHH07] remain valid in the sample that I use.

Last, we can observe that in the last years, the topic has shifted from validation ([GT00, RAGBH05, Koc05]) to modelling ([Wu06, WHH07, HGBR07b, CGBHR07, CFR07]). I also address that question in this thesis. I study the statistical properties of software, to test whether previous empirical findings hold for the set of case studies that I use. Furthermore, based on those properties, I propose to apply some models that have been proved valid in other fields that share the same statistical properties [Mit04b].

Epoch	Topic					# case studies		
	Validation	Statistical model	Physical model	Blended	Other	$1-5$	$5-15$	> 15
70s-80s	[Pir88], [Law82]	[Leh74], [Cho88]	[BL71], [Leh74], [BL76], [Woo80]		[Pir88], [Leh74]	[Leh74]	[Law82]	
90s	[LR02], [LRW+97], [GJKT97], [AP01]	[KS99]	[Tur96]	[AP01], [KS99]	[RB00]			[KS99]
Libre software	[GT00], [GT01], [CMR04b], [RAGBH05], [Koc05], [HRGB+06]	[CMR04a], [GT00], [RAGBH05], [Koc05], [HRGB+06], [CLM03], [PLM07], [ACPM01], [CCPV01]	[Tur02], [RMGB05], [ASAB02], [ASS+04]	[WHH07], [Wu06], [CFR07], [HGBR07b]	[CGBHR07]	[GT00], [GT01], [PLM07], [CFR07]	[HRGB+06], [CMR04a], [Wu06], [WHH07]	[RAGBH05], [Koc05], [CLM03], [HGBR07b]

Table 2.7: *Classification of empirical studies on software evolution*

CHAPTER
THREE

Methodology

3.1 Introduction

This chapter describes the methodology used to answer the different research questions raised in chapter 1. For each one of those questions, the methodology has two main stages: collection of data, and analysis. For the first stage, collection of data, I have used exclusively public data sources, to ensure the reproducibility of this work. For the second stage, as mentioned above, I have used a statistical approach, using different techniques.

More in detail, we can classify those research questions in three groups, classifying by the different techniques used to address them:

- Questions addressed using a correlation analysis:
  - Are the laws of software evolution valid for the case of libre software?
  - Which and how many metrics should be used to characterize a software product?
  - Which is the shape of the statistical distribution of the size of software?
  - How does that shape vary with the scale of the metric?
- Questions addressed studying the dynamics of evolution using time series analysis:
  - Which kind of dynamics is driving software evolution?
- Questions addressed obtaining models for the evolution based on time series analysis:
  - How can we use that information about the dynamics to model and forecast software evolution?

For the first group, I have used a sample of thousands of software products obtained using the FreeBSD packages system. I measured different size and complexity metrics, and correlated them to answer some of the research questions mentioned above: Are studies using different metrics comparable? Which are the metrics that we need to characterize the size and complexity of a system? I used the same dataset to determine which are statistical properties of the size of software (shape of the statistical distribution, how it varies with different metrics, etc).

For the second group, I have used a set of software projects obtained from the FLOSSMole and CVSAnalY-SF datasets. That is because the dataset mentioned in the previous paragraph lacked the historical information needed to perform a time series analysis.

For the third group, I have used other three different case studies. That is because I needed to have a deep knowledge of the case studies, to overcome possible flaws for the models that I obtain. In any case, although this analysis has not been done at a large scale (contrarily to the two mentioned above), it addresses the question about how to obtain accurate statistical models for the evolution of software.

The rest of this chapter is as follows. Section 3.2 describes the correlation analysis, the procedures used to obtain the sample for it, and the different metrics considered for this study. Section 3.3 describes how I have applied time series analysis to determine which kind of dynamics drives software evolution, including the procedure followed to gather the sample with historical information. Finally, section 3.4 shows how to apply time series analysis to forecast software evolution, based on the information determined by the methods shown in section 3.3.

3.2 Correlation analysis

In section 1.3.1, among others, I raised the following research questions: Are the laws of software evolution valid for the case of libre software? Are different empirical studies comparable if they are using different metrics? Which attributes better describe a software product? Which are the statistical properties of those attributes?

To address those questions, I have used a correlation analysis, that I have performed over a sample of thousand of software products obtained using the FreeBSD packages system. For all those products I measured size and complexity, using different metrics. Among all the possible, I have selected those metrics that are commonly found in the empirical works of software evolution. This approach will make it possible to determine whether or not the metrics used in those studies are comparable, and thus it will make it possible to determine if it is possible compare studies that use different metrics.

This section describes in more detail those two issues: collection of case studies and their metrics, and correlation analysis.

3.2.1 Data sources

The data sources used to obtain the sample for this study must fulfill the requirement of being public, automatically downloadable, and include a large number of software products. Furthermore, the source must be unbiased in the sense that it must contain different domains of application. An additional requirement is that the products included in the source must be libre software.

All these requirements are related to the main goal of this thesis, shown in section 1.3.1. I need to obtain a statistically significant amount of case studies, this is, a large number of software products, which can only be studied in an effective way by means of automatic methods. The results of my study must be repeatable by anyone, which needs the data to be public. Furthermore, the scope of this thesis is bounded to the case of libre software.

From the different alternatives, software distributions and packages repositories are one that fulfills all these requirements One of the most well known software distributions is FreeBSD, which as Linux can be considered a flavor of Unix.

I decided to use FreeBSD as source of all the software packages to be studied because it fulfills all the requirements mentioned, although some of the software packages included in FreeBSD can not be redistributed, and therefore are not libre. However, all those packages were excluded from this study.

In FreeBSD argot, software packages are called *ports*. Ports include source code for applications or libraries. They contain, among other files, a *makefile*, which is able to download the source code from the original web or FTP site, to apply patches developed by the *porters* of FreeBSD (if any), to compile, and to install it into a live system. It includes the following files

- `Makefile`
  It includes information about the site where the source code can be downloaded from, and about how and where the program will be installed in the system.

- `distinfo`
  It contains checksums for the files that the port downloads. With those checksums, the ports system check the integrity of the downloaded files.

- `pkg-descr`
  It contains a description of the program.

- `pkg-plist`
  It includes a list of the files that will be installed.

Sections are another property of ports. They mark where the port is stored within the hierarchy of the tree of ports. Moreover, sections give information about the domain of application to which the port belongs.

I have used the `Makefile` and the `distinfo` files to obtain the source code. The `Makefile` contains two variables called `MASTER_SITE` and `MASTER_SITE_SUBDIR`. Together with the list of files included in `distinfo`, the package system can build a URL that is used to download the sources of the program from the original site. This is done using `make fetch` in the directory where the port files are stored. The sources are typically stored in the `/usr/ports/distfiles/` directory.

I invoked the command in a FreeBSD 6.2-RELEASE system, running on an AMD64 platform. At the time of the study (May 29th 2007), the collection contained 16,253 ports. From that collection, I included in the sample only 12,108. After uncompressing all the sources corresponding to those 12,108 ports, I obtained a set of 1,446,209 files. Those ports were classified in 62 different sections.

3.2.2 Validation of the sample

Of the total number of files, 591,821 were written in the C programming language (being 202,968 header files, and the rest 388,853 non-header files). Only 6,665 of the 12,108 ports contained source code written in C. Those ports written in C belonged to 60 sections. These statistics are summarized in the first column of table 3.1, which shows the basic properties of the sample after each one of the validation steps.

For a correlation analysis to be significant, the sample must fulfill some requirements. The first requirement is that every point that is going to be part of the correlation must be independently generated. For the case of source code files, the first obvious filter is to remove items that have not been independently generated. I also removed all the version of each port but the last, because the files that are part of different versions of the same port are likely to be statistically dependent.

Besides repeated files, I also removed all the files that were generated by other tools and not written by people. I also found problems trying to measure some of the files, and decided to remove them from the sample.

More details about these validation steps are given below. A summary of the contents of the sample after each step is shown in table 3.1. Full details about the statistical properties of the sample are given in table 4.3.

Repeated files

It is obvious that duplicated files have not been independently generated. I used the MD5 hash function to identify them.

FreeBSD contains several different versions of the same programs and libraries. For instance, different versions of the programming language Python are included because of compatibility issues. I considered the latest version of each port and I discarded the rest. The source code of different versions was actually different in most of the cases. However, those files have not been independently generated. It is quite likely that the files in the newer versions are based on the files in the previous versions, and although not exactly duplicated, newer files will contain chunks of source code that comes from older versions.

		Retrieved sample	**Non-repeated**	**Non-automated**	**Final sample**
All programming languages	*# Files*	1,446,209	1,109,697	–	–
	# Ports	12,108	12,010	–	–
	# Categories	62	61	–	–
Only C source code	*# Files*	591,821	459,372	449,707	447,612
	# Headers	202,968	158,826	154,161	152,937
	# Non-Headers	388,853	300,510	295,546	294,675
	# Ports	6,665	6,587	6,556	6,556
	# Categories	60	60	60	60

Table 3.1: *Summary of the validation process of the sample*

I found 336,512 files that were either duplicated or belonging to different versions of the same port. After removing those files, the sample contained 1,109,697 files, belonging to 12,010 ports distributed in 61 categories. From that sample, I extracted all those files written in C. The set contained 459,372 files written in C, being 158,862 header files and 300,510 non-header files.

Automatically generated files

In the libre software world, the use of lexical analyzers is very popular. Those tools usually return C code. I decided to remove those files, because I wanted to select source code written by humans.

To identify automated files I used regular expression matching, using the first 50 lines of the files. Flex and similar tools include a comment at the beginning of the files, that indicates the file was generated by another tool. I used 19 different patterns, shown in appendix D.

With this procedure I identified 9,665 files that were automatically generated. After removing them, the sample contained 449,707 files, being 154,164 header files and 295,546 non-header files. The resulting files belonged to 6,556 ports, distributed in 60 categories.

Unparseable files and dummy headers

The measurement tools that I used could not correctly measure 2,013 files. I also found 82 files with 0 SLOC. After inspection of those 82 files, most of them were header files that contained comments that explained that those *dummy* header files were necessary to fix the compilation process in some platforms. I removed all of them from the sample.

Therefore, the final sample contained 447,612 files, being 152,397 header files, and 294,675 non-header files. The files belonged to 6,556 ports distributed in 60 categories.

3.2.3 Selected metrics

Size and complexity can be measured using many different metrics. Some of the metrics are computed using language dependent algorithms. Therefore, in order to measure them, the most representative languages must be selected.

Although C is not the most popular programming language nowadays, it is the most used in libre software systems. As a matter of fact, the 4.0 release (the latest at the time of writing this) of the Debian GNU/Linux distribution contains 145.2 millions of SLOC written in C out of a total size 288 millions [RGBM$^+$08]. This is 51.3% of the source code. Therefore, more than half of Debian is written in C. In the case of the version of FreeBSD studied in this chapter, it contains 141 millions of SLOC out of 281, which amounts to 50.4% of the source code.

We can consider FreeBSD (or Debian) as a proxy of the overall libre software. If a software system is popular enough, it gets included in FreeBSD. Assuming that, the sizes found in the collection of software are lower bounds for the real sizes corresponding to the whole collection of libre software.

Therefore I can assume that about half of the code present in the most popular systems of the libre software community is written in C language, and I can bound this study only to that language without losing representativeness. Focusing in the case of the C programming language, I have collected the following metrics:

Source Lines of Code (SLOC)

I use the definition given by Conte [Con86]:

> A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements

This metric was used in the first studies that highlighted that libre software was not evolving as the Lehman's laws predicted [GT00].

Lines of Code (LOC)

I measured the number of lines of a source code file, including comments, blank lines, etc. I used the Unix tool *wc* to measure this metric.

Number of C functions

I counted the number of functions inside each file. To count functions, I used the tool *exuberant-ctags*, combined with *wc*.

McCabe's cyclomatic complexity

I use the definition given by McCabe [McC76], which indicates the number of regions in a graph. For a graph G with n vertices, e edges, and p exit points (e.g., function returns in the case of C), the complexity v is defined as follows:

$$v(G) = e - n + 2p \tag{3.1}$$

Figure 3.1 contains a sample graph, that may represent the different execution paths of a program. Any graph has at least one region (the surrounding region). Therefore, the minimum value of the cyclomatic complexity of a program is 1. Any program can be easily represented as a graph, with simple rules such as representing `IF` statements by bifurcation points, or closed loops corresponding to regions, etc.

The rationale behind the metric is that all the edges that delimit a region form a *circuit* that must be tested. The more regions a program has, the more testing it will need, and the more complex the program is.

Because of that, for the case of the C programming language, this metric is defined at the function level. To obtain its value at the file level, I calculated the cyclomatic complexity for every function within each file, and aggregated those values to obtain the complexity at the file level.

Halstead's Software Science metrics

I used the definitions given by Kan [Kan03]. I have measured four metrics: length, volume, level and mental discriminations. These metrics are based on the redundancy of *operands* and *operators* in a program.

For the case of C, operators are string constants and variable names, and operands are symbols (like `+`,`-`,`++`,`-`), the `*` indirection, the `sizeof` operator, preprocessor constants, statements (like `if`, `while`), storage class specifiers (like `static`, `extern`), type specifiers (like `int`, `char`) and structures specifiers (`struct` and `union`).

The metrics are obtained by counting the number of distinct operators n_1, number of distinct operands n_2, total number of operators N_1, and total number of operands N_2. The length L of a program is the total number of operators and operands:

$$L = N_1 + N_2 \tag{3.2}$$

The volume V of a program is defined as

$$V = N \cdot \log_2(n_1 + n_2) \tag{3.3}$$

The level lv of a program is defined as:

$$lv = \frac{2}{n_1}\frac{n_2}{N_2} \tag{3.4}$$

The inverse of this metric is sometimes mentioned as *difficulty*.

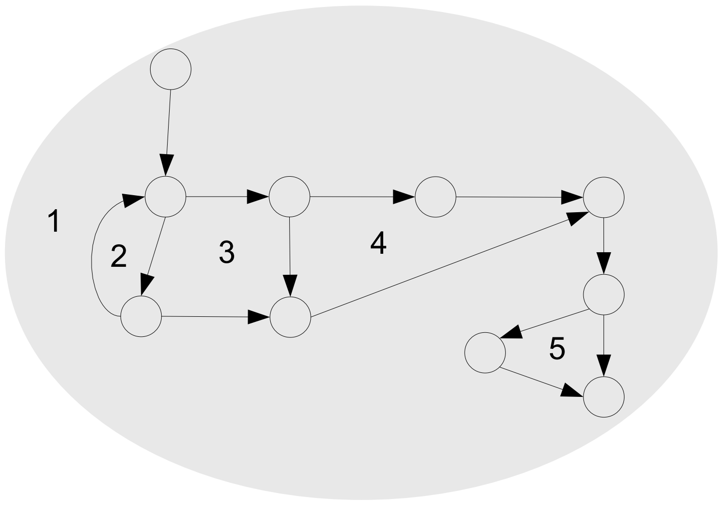

Figure 3.1: Sample graph that represents the flow of a program. The cyclomatic complexity is the number of regions in the graph. Any graph has at least one region (the surrounding region). So the minimum value of the cyclomatic complexity is 1*. In the graph shown in this figure, the value of the cyclomatic complexity is* 5*.*

The effort E that a programmer needs to invest to comprehend the program is defined as

$$E = \frac{V}{lv} \tag{3.5}$$

This metric is sometimes denominated *number of mental discriminations* that a developer must do to understand a program.

All the mentioned metrics have been collected at the file level[1]. The statistical analysis described here has been done at two levels: file level and aggregated level. The reason is that the controversial studies mentioned in the research question about the comparability of software evolution studies (mainly the classical works by Lehman summarized in the 1985 book [LB85], and the paper by Godfrey and Tu about the growth of Linux [GT00]) are done at aggregated levels. Lehman uses number of files (or modules, in the terminology that he uses), and Godfrey number of aggregated SLOC (at the project level).

Table 3.2 shows the selected metrics, and the symbols to denominate each metric in the rest of this thesis. I have used the following tools to measure those metrics:

- **SlocCount**
  SlocCount is an integrated set of tools to measure SLOC for a myriad of programming languages. Besides measuring SLOC, the tool uses some heuristics to identify the programming language that a file is written in. Furthermore, it contains heuristics to determine if a file is of automated origin or has been written by a human. Those heuristics resulted to be insufficient though, and I had

[1]With the exceptions mentioned in the case of the McCabe's cyclomatic complexity

Attribute	Metrics
Size	Source Lines of Code (SLOC), Lines of Code (LOC), Number of C functions (FUNCS)
Complexity	McCabe's cyclomatic complexity (CYCLO) Halstead's length (HLENG), Halstead's volume (HVOLU) Halstead's level (HLEVE), Halstead's mental discriminations (HMD)

Table 3.2: *Selected metrics to measure size and complexity.*

to extend them for this study. SlocCount is libre software can be obtained at http://www.dwheeler.com/sloccount/.

- **Libresoft's CMetrics**
  CMetrics is a set of tools to measure different complexity metrics for C source code files. Among those metrics, we can find all the complexity metrics shown above. CMetrics is libre software and can be obtained at http://tools.libresoft.es/cmetrics/.

- **Unix's `wc` and `exuberant-ctags`**
  In order to measure lines of code, I used the `wc` command, that measures the number of lines in a text file. To measure the number of functions I used the `exuberant-ctags`. That tool support many different programming languages and can extract different entities from source code files. I used it to extract the functions from C source code files, and then I counted the number of functions using `wc` (`exuberant-ctags` write the name of each function in a separated line, and `wc` can count the number of lines). Both tools are available in any typical installation of FreeBSD (the system used for this study).

3.2.4 Statistical analysis

The correlation analysis was done at the file level, pairwise. This is, for each file I obtained eight values corresponding to each one of the metrics. I took each file as an independent point, and performed a correlation analysis for each pair of metrics.

This procedure was done with the whole sample. To test the sensitivity of the correlations to different properties of the sample, I repeated the analysis using only some subsamples. The overall sample is heterogeneous, including packages with very diverse application domains, both header and non-header files, etc. I split the overall sample in more homogeneous and smaller subsamples. Figure 3.2 illustrates this breakdown process. The classification criteria that I used are the following:

- **Type of file**
  I divided the final sample in two subsamples, one containing exclusively header files and the other one non header files. I analyzed how the correlations varied in each subsample when compared to the overall correlation coefficients.

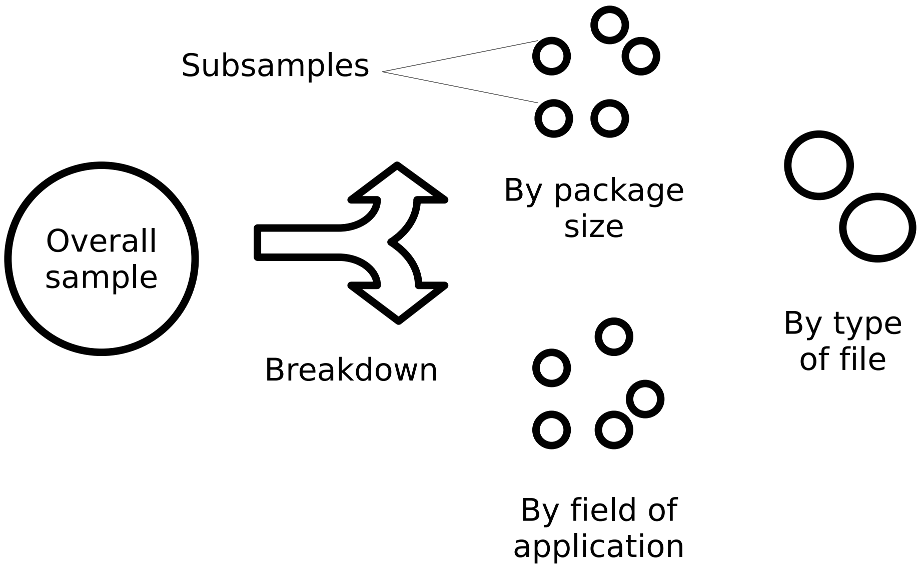

Figure 3.2: Breakdown process. The statistical analysis was repeated in subsamples of the overall sample, to test the sensitivity of the results with three different parameters: filetype, package size and field of application. This process helps to determine if the results are due to aggregation effects and therefore are not significant.

- **Package size**
  The breakdown by package size was done in two different ways:
  - Across packages. The range of package sizes was divided in eleven intervals, and the files in each interval formed each one of the subsamples. I analyzed how the correlation coefficient varied in each interval.
  - Within packages. I calculated the correlation coefficient using the files in each package. I analyzed later how the correlation coefficient was related to the package size, to find out if it varies with that parameter.
- **Field of application**
  There were 60 different categories in the sample. I divided the sample in 60 subsamples, one for each category. Then the correlation analysis was done for each one of those subsamples, to determine how the correlation varied with the category.

Finally, I did the analysis for the aggregated attributes at the package level. This is, for each package, I calculated the properties for the package as the addition of all the files included in that package, and I repeated the same analysis using those aggregated values. This provided an additional metric: number of files in the package, which is the same used in the classical studies of software evolution (see chapter 2). Lehman has argued that aggregated attributes (like number of files) should be used instead of low level metrics (like SLOC) [LPR98]. This issue is described in detail in section 2.3.4. I repeated the same analysis with the additional metric of number of files, to determine if there was any relationship between number of files and the rest of metrics, and empirically test the problem highlighted by Lehman, which is one of the research questions that I mentioned in section 1.3.1.

3.3 The dynamics of software evolution

In 1974, Lehman proposed to use a statistical approach to study the evolution of software [Leh74]. He specifically mentioned time series analysis as one of the techniques to be applied to the case of software evolution. Based on that, one of the research questions addressed in this thesis is about how to apply a statistical approach to study software evolution, and in particular, about how to use time series analysis to obtain a statistical model for evolution. The methodology used to address both questions is shown in this and the following sections. This section shows how to characterize the dynamics of software evolution using time series analysis. To obtain a model for evolution using time series analysis, we need to know which kind of process we are trying to model. This is, depending on the statistical properties of the evolution of software, the terms and parameters of the model will be different. Therefore next section will show how to use that information to obtain a statistical model of software evolution, using again time series analysis.

In order to obtain meaningful results, this approach requires the sample to be statistically significant. In the analysis shown in section 3.2, I have shown how to use the ports system of FreeBSD to obtain a large sample of software systems. However, that sample lacks historical information. Software evolution studies (and time series analysis) need historical information. Therefore, I used a different sample in this case. I obtained a statistically significant sample, with enough historical information, using the FLOSSMole [HCC06] and CVSAnalY-SF datasets.

3.3.1 Data sources

As in the case of correlation analysis, I obtained the sample from a public data source that exclusively contains libre software. The size of the sample should large be enough to be statistically significant. Moreover, for time series analysis, the sources must contain historical information, and the period covered has to be long enough to apply time series analysis.

In section 1.2.2, I described the data sources available in a typical libre software project. From all of them, source code management systems are the most interesting for time series analysis, because they contain the whole history of changes for a software project. Each record contains who made the change, when, where (in the source code tree), and probably why (in a comment).

The problem is to obtain a large amount of control version repositories. To overcome this difficulty, we used a *forge*. Forges are sites that offer hosting for libre projects, and provide the most usual tools that are necessary for the development processes. The most popular in the libre software community is SourceForge.net (SF.net).

All the libre software projects hosted in SF.net share a common structure. So it is easy to extract information in an automatic way. Because of that, it has become popular in the research community as well. In particular, the FLOSSMole research project was launched to mine information in SF.net (and other forges), and to offer the datasets to the research community.

FLOSSMole parses the web pages of SF.net, and creates a database including information regarding all the projects hosted in this site. The information is meta-data such as the licenses, number of developers, number of releases, etc. It is starting to gather some other data, such as mailing lists information, but it still lacks change records.

The FLOSSMole database was used by our research group to create a database containing the whole change history for all the projects that have a CVS repository. The CVSAnalY-SF dataset [2] contains a register for each commit in the CVS repository of all of the projects. Each register contains who made the change, the date of the change, the path of the file affected by the change, and the log text written by the developer to give some information about the change.

Selection criteria

Many of the projects hosted in SF.net are pet projects with very low activity. Many of those remain inactive for long periods of time, in some cases forever. That is mainly because they do not manage to attract new people to the project. Therefore, the size of the community around of those projects is very small.[3]

The CVSAnalY-SF dataset contains change records for $29,839$ projects, as of June 2006 (the latest available version at the time of writing this). But it is likely that not all of those projects are active, and if they are active, they can be too young as to be of interest for this study.

To distinguish whether a project was active or not, I used the criterion proposed by Capiluppi and Michlmayr [CM07]. These authors propose that projects with very few developers are probably still in an early stage in their history. Their research has shown that the behaviors of projects at different stages of evolution are different. They find three stages in the evolution of projects. The transition from one phase to another is gained when it achieves to attract more developers. The first phase in the history of the project is called by Capiluppi and Michlmayr the *cathedral phase*. The next phase is a transition phase to the third, that is called the *bazaar phase*. These labels are used to match the terms used by Raymond in his paper about the different dynamics of proprietary and libre software [Ray98].

Summarizing, and using the terminology of the mentioned paper, I wanted to select all those projects likely to be in the bazaar phase. The empirical criterion that I used

[2]Available at http://libresoft.es/Results/CVSAnalY_SF

[3]In fact, more than 82% of the projects stored in the FLOSSMole database as of June 2007 include 2 or less developers.

was to select all the projects with at least 3 developers. I used the value stored in the database of FLOSSMole, explained in the previous section and shown in table 3.3.

The second requirement is having at least one year of history. I wanted to compare my results to those reported by Wu [Wu06, WHH07], who studies a year of changes extracted from the whole history of the projects. Therefore, I selected only projects with at least 12 months of history in the CVS repository.

Properties of the sample

After applying the different selection criteria described above, the sample contained 3, 821 projects. For each one of those projects, I obtained the daily number of changes, considering only source code files. I identified changes corresponding to source code thanks to the information contained in the LibreSoft's dataset. The tool used to obtain that dataset, CVSAnalY, identifies the kind of file that was changed in each commit, and marks that change in the database with a determined type (documentation, images, translation files, source code, etc). Thus, for each one of the 3, 821 projects I had a time series of the daily number of changes.

I also obtained the *modification requests* (MRs). In CVS, commits are done at the file level. If a change affects different files, it will appear as more than one commit in the CVS repository. In other words, a change or modification request in the sources tree will appear in the version control repository as more than one change, because of the limitations of CVS. To reconstruct the modification requests, I used the algorithm described by Germán [Ger04].

Table 3.3 shows the properties of the sample of selected projects. The number of developers was obtained using the FLOSSMole database. The project page in SF.net contains a field indicating the number of developers that work in the project. That field is increased each time a new developer joins the project. It is not related to the activity in the CVS. Although joining a project is a requisite to get access to the CVS, being a developer in the project in SF.net does not imply any participation in the CVS. In other words, that number is an upper bound for the actual number of developers working in the CVS.

Regarding the values labeled as *SF.net age* and *CVS age*, those columns indicate the age of the projects in months. The first value is the number of months that the project has been stored in SF.net. The second value is the difference in months between the dates of the last and first commit in the CVS. Because not all the projects start to use CVS right from the beginning, SF.net age is an upper bound for CVS age. The age of the projects was measured using the CVS age. Thus, I removed from the sample all those projects with less than one year of CVS age.

The next two columns (SLOC and number of files) indicate the size of the project in SLOC and the number of source code files. Those values were obtained using SlocCount. The source code measured corresponds to the latest checkout available in the CVS repositories in February 2008. The change data obtained from the CVSAnalY-SF

	# Devs.	SF.net age	CVS age	SLOC	# Files	# Changes	# MRs
Min.	3	29	12	10	1	2	1
Max.	354	92	91	12,028,586	44,174	126,354	34,480
Mean	8	64	33	72,903	374	2,417	619
Median	5	64	29	21,168	142	850	206
Sd. dev.	12	17	17	341,354	1,165	5,626	1,477

Table 3.3: *Statistical properties of the sample of 3821 projects. # Devs. are number of developers. SF.net age indicates the number of months that the project has existed in SF.net. CVS age indicates the number of months that have elapsed between the first and the last commits in the CVS repository. SLOC measures size in Source Lines of Code (it excludes blank and comment lines). #MRs is number of modification requests.*

dataset was recorded in June 2006. The last two columns indicate the number of commits that have occurred in the CVS repository, and the number of modification requests.

3.3.2 Statistical analysis

The first question when applying time series analysis to software evolution, is to determine which kind of dynamics drives software evolution. That will help to choose the appropriate model for software evolution.

One of the main properties of time series is the autocorrelation coefficients. These are the linear correlation coefficient among the time series and the series itself, but shifted one position to the future. Thus, we may obtain up to $n-1$ coefficients, where n is the number of *lags* of the series. For instance, if the time series was collected daily, each day will be a lag. Figure 3.3 shows a diagram that explains how the coefficients are calculated. For instance, $r(1)$ is calculated correlating the original series against the series shifted one position, $r(2)$ is the same but the series is shifted two positions. Following this procedure iteratively, we obtain $n-1$ coefficients, being n the number of points of the original time series (also called lags, as mentioned above).

The autocorrelation coefficients function (ACF) measure the linear predictability of the series, using only values of the past of the same series. What is more important, the shape of the plot of the autocorrelation coefficients gives us information about the kind of dynamics of the process that we are studying. The shape of the plot can also be used to calculate the parameters of a time series model.

Figure 3.4 shows the *theoretical* profiles for two extreme case of dynamics: short memory, and long memory processes. Long memory processes present a slow decay of the autocorrelation coefficients. Equation 3.6 shows the relationship among autocorrelation coefficients $r(k)$ and time lags k for an ideal long term process.

$$\log r(k) \sim C + (2d-1)\log k \tag{3.6}$$

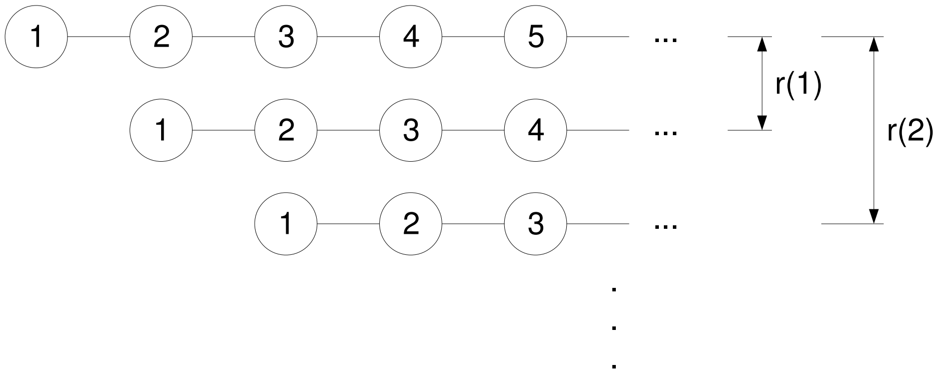

Figure 3.3: *Autocorrelation coefficients in a time series. The coefficients are calculated correlating the series against the same series but shifted one position. If the series has n elements, there are $n - 1$ coefficients. In the plot, each circle represents a point in the series. The plot shows how the series is progressively shifted one position, and how each coefficient is obtaining by linear correlations of the original series and its shiftings.*

being C a constant and d one of the parameters of the model of the process (it is related to the memory of the process).

A process is said to be long memory (or long term, or long autocorrelated) when $0 < d < 0.5$. The ideal long memory process presents a linear profile in the logarithmic plot of autocorrelation coefficients against time lags, having the slope a particular value in the interval shown above.

In the other hand, short memory processes present a fast decay of the coefficients, being the ideal process a linear relationship among the coefficients and the lags[4]. Equation 3.7 shows the linear relationship among autocorrelation coefficients $r(k)$ and time lags k for an ideal short memory process.

$$r(k) \sim C\,(1 - \epsilon \cdot k) \tag{3.7}$$

where ϵ and C are constants.

The typical example of a short memory process is the stock market, where the value of the index today depends at most on values of the index during the last days, but very old results of the index do not affect its current value. A real process would be somewhere among those extreme profiles.

Another similar function is the partial autocorrelation coefficients function (PACF). That function does not provide information about the kind of process, but it is needed to obtain the parameters for a time series model[5].

[4]Actually, that linear profile correspond to a particular type, denominated ARIMA integrated processes, which I will review in the next section.

[5]Appendix C gives more details about the functions and equations described in the last paragraphs.

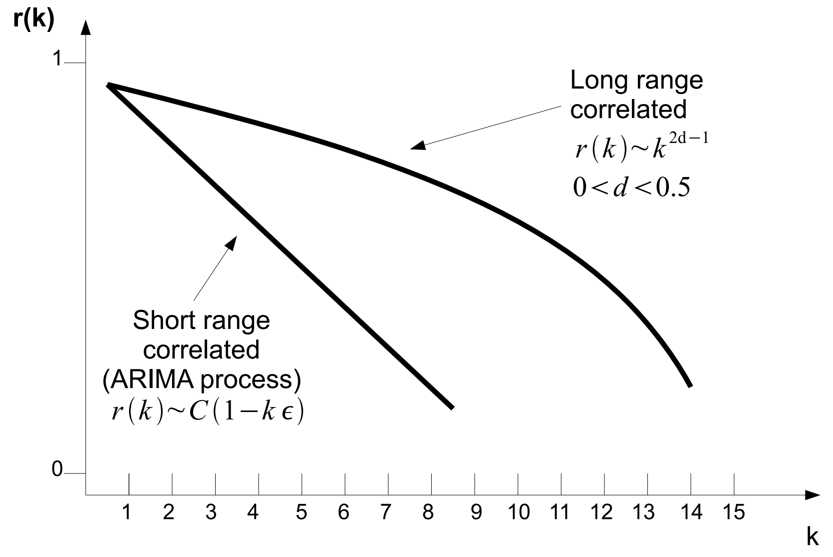

Figure 3.4: *Theoretical profiles of the autocorrelation coefficients of long term and short term processes. This diagram shows the mathematical relationship among the coefficients and the time lags.*

If the empirical data contains noise, the ACF and PACF do not show a clear pattern, and we cannot apply the analysis shown in this section. In order to remove the noise of the samples, I applied *kernel smoothing*, as described by Shumway and Stoffer [SS06]. This *kernel smoothing* filter makes the series smoother by introducing some autocorrelation in the data. After the smoothing process, the pattern of the ACF and PACF is more clear. If too much smoothing is done, the data will artificially show a pattern due to the autocorrelation added to the data. In the chapter 4, I discuss the influence of the smoothing process in the validity of the results.

Summarizing, I obtained the daily time series of changes and modification requests for all the $3,821$ projects of the sample. I obtained the ACF plots for each one of those projects, and I applied kernel smoothing when necessary to obtain a clear pattern for the ACF plot. After those steps, I matched the plot against the two extreme cases described above, to find out whether each project was driven by a short or long memory dynamics.

To determine this last issue, I computed a linear correlation of the autocorrelation coefficients against the time lag. If the ACF plot is linear, the process is driven by a short memory dynamics. Otherwise, it is not. Thus, I obtained a set of $3,821$ linear correlation coefficients. I calculated an estimation of the density probability function, to quantify how many of the projects could be considered driven by a short memory, and how many by a long memory.

3.4 Forecasting software evolution

Last section has shown how to use time series analysis to characterize the kind of dynamics that drives software evolution. That information is useful to obtain a statistical model of the evolution of a software system. I show in this section how to obtain those models, using time series analysis.

The sample used for this analysis is different that the sample used in the previous section. The reason is that the requirements for the sample are different. While in the previous section the goal was to obtain a sample broad enough to get statistical significance, in this section the goal is to obtain more detailed information about each case study, because that information is fundamental to obtain proper models for their evolution. For instance, as I show later in this section, I have removed some files from the projects that I have chosen as case studies, because those files could not be actually considered part of the systems.

Right after describing the selection criteria and procedure for the sample, I show how to apply time series analysis to obtain a predictive model for the evolution of the system.

3.4.1 Data sources

As in the previous sections, the main requirement for the data source for this analysis is that all the data must be publicly available, and that the software systems must be libre. In the case of modelling, there are two additional requirements. First, the case studies must have enough historical information for the fitting procedure of the model. Second, the case studies must provide contextual information, to get a deep insight in each one of them, to detect anomalies that could be flaws for the models.

The parameter that I forecasted is size. Therefore, an additional requirement is to have access to the source code of the project, over its whole history, and with the proper resolution. This is, to forecast the daily series of size, we need to know the size of the system at each day since the beginning of the project.

Because of all these requirements, I could not reuse all the cases that I used in the previous section. I decided to use three case studies that fulfilled all the requirements previously mentioned, as well as that I knew with some degree of detail. In particular, all the cases that I finally selected offered their control version repositories (CVS in all the cases) to be mirrored, what greatly simplified the task of obtaining the daily series of source code size.

The three case studies that I selected were:

- **FreeBSD kernel**
  This system has been already described in section 3.2.1. But the focus is different here: I am interested in the development and evolution of the FreeBSD kernel. From the CVS repository, I selected the module `src/sys`, and all the commits that were in the main trunk of the repository.

- **NetBSD kernel**
  NetBSD is another flavor of the BSD family of operating systems. It was the second libre software BSD variant to be formally released, after 386BSD, and continues to be actively developed. From the CVS repository of this project, I selected the module `src/sys`, and all the commits that were in the main trunk of the repository.

- **PostgreSQL database management system**
  PostgreSQL is an object-oriented relational database management system. Its origins are in 1970s in the University of California at Berkeley, in the Ingres project. Not much code of Ingres remain nowadays in PostgreSQL though. From the CVS repository I selected the `src` module, and all the commits within that module in the main trunk.

Data collection

I used the softChange tool [GH06] to analyze the CVS repositories of the projects, and to obtain a database with all the changes history of the project. A change stored in CVS (a revision, in the CVS argot) can be understood as a point of change. Every time a change is committed to the CVS, a new revision is created. The database obtained using softChange contained a table with all the information needed to identify the revision. In particular, from this database I read the revision identification tag, the filename, a flag to determine whether or not the revision is in the main trunk, a flag to determine whether or not the file was removed in that revision, and the date and time for that revision.

With the filename and the revision identification tag, I could obtain the original file from the CVS. With the help of the flags, I could retrieve only files in the main trunk (avoiding the files in parallel branches) and avoid revisions where the files were deleted. Finally, the date and time allowed me to sort all the revisions in a chronological order.

With all the information, I could reconstruct the characteristics of the source code of the software product just as it was at the moment of each revision. For the case studies mentioned above, I did that for all the files written in the C programming language, ignoring header files. I only considered all those files that were in the main trunk of the repositories. This information is at the revision level. This is, it lacks date and time as to sort the different versions in a chronological fashion.

I could convert that data into natural time with the following procedure. For each one of the files, I retrieved all the revisions for every day. If a file had more than one revision in the given day, I retrieved only the last one (given by the hour filed). I built a tree of the sources, initially empty. Then I applied the same changes to this tree than those that were made to real source code tree. Every leaf in this tree were not files, but meta-data containing the size of the file and the information needed to identify the file (revision id and file path). Therefore, for every day, I had a tree containing the sources of the project as they were in the considered day. Hence, instead of the actual files, I had the values of their size. For every day I aggregated the values of all the files. With this procedure, I obtained a time series with the daily size of each one of the three case studies.

Furthermore, using this approach the time resolution is arbitrary. I selected one day as the period for the time series. But I could obtain time series with a resolution as short as one hour, or as wide as one year. This revision approach allows to obtain time series of metrics with an arbitrary period, avoiding to measure each version of the file more than once.

The other typical procedure to measure size over time consists of taking snapshots at certain moments of the history, and measure them. However, by measuring snapshots we would measure many times all the files that have not changed, which is very inefficient. If the period is small, the amount of repeated measurements would be large. Moreover, the period of the snapshots will be the minimum period for the time series, so loosing resolution in the study of the evolution. By measuring revisions, we only measure the points of change of the files, and we do not loose any resolution in the time series.

Therefore, using the procedures explained in the last paragraphs, I obtained a daily time series of the size of each one of the three case studies. I used those series (and their properties) to obtain a predictive model for the evolution of the three systems.

Data filtering

I observed some odd jumps in some of the series, and decided to explore further to find out the reason of those sudden changes. In particular, for FreeBSD I observed "steps" in the growth curve on May 28th 2000, on May 29th 2001, and on December 12th 2005. The first jump (May 28th 2000) corresponded to the addition of the module `sys/dev/acpica`. The second jump (May 29th 2001) corresponded to the removal of the module `sys/contrib/dev/acpica/Subsystem`, that was old code for ACPI support in FreeBSD and was no longer being used. The third jump (December 12th 2005) corresponded to the addition of the XFS filesystem (limited to read-only support). However, after careful inspection of all these jumps, it seemed that everything was the result of the development process itself. Therefore, I decided not to remove any of the modules from the case study.

However, I did remove some modules in the case of PostgreSQL. In the CVS repository of the project, during some time, two files were removed and added many times. One developer was removing them, claiming that those files were produced by the build system. The other developer added the files each time she made new changes to that module. The files affected were `src/interfaces/ecpg/preproc/preproc.c` (with $2,758$ SLOC in its latest version) and `src/interfaces/ecpg/preproc/pgc.c` (with $16,473$ SLOC in its latest version). These two files were not written by humans, but were the output of the YACC parser generator. Because of that reason, I decided to remove those two files from the sample.

Also in PostgreSQL, I found that the `src/interfaces/odbc` module was spun off at some point. That module began being part of the repository of PostgreSQL, but the

	$q = 0$	$p = 0$	$p \neq 0$ $q \neq 0$
ACF	Tails off	q significant coefficients	Tails off
PACF	p significant coefficients	Tails off	Tails off

Table 3.4: *Criteria to select the values of p and q in an ARIMA model. ACF means autocorrelation function. PACF means partial autocorrelation function. Reproduced from [SS06].*

development team was completely different. After some time, that team decided to move to their own repository, causing a big negative gap in the growth curve of the PostgreSQL's CVS repository. Therefore, I decided to remove that module from this study.

3.4.2 Statistical analysis

Because of internal autocorrelation, time series can be predicted based on past values. The degree of internal autocorrelation in the data is measured by the autocorrelation coefficients and the partial autocorrelation coefficients.

If the process that we want to forecast is driven by a short memory dynamics, we can use ARIMA (Auto-Regressive, Integrated, Moving-Average) models[6]. ARIMA models are linear combinations of past values of the series, weighted by some coefficients. To obtain those coefficients we need to identify three different parameters: p, d and q. A seasonal component can be added to the model. For instance, if we would know that any of the projects is releasing a new version with a stable period (for instance, a new release every six months), we could aggregate that information to the model, so increasing the goodness of the prediction. However, I have not tried to add a seasonal component to the models in the present study.

The values for the parameters are obtained using the Box-Jenkins method [MWH98]. The first requirement to apply this method is that the data needs to be stationary. The number of differences that must be applied to the data is the value of the parameter d. If the series are close to linear, then stationary data can be obtained applying the first difference ($d = 1$).

In order to obtain the values of p and q, we must inspect the plot of the autocorrelation and partial autocorrelation coefficients. The criteria to choose the values of p and q are explained in table 3.4. To apply those criteria, we have to plot both the autocorrelation coefficients function, and the partial autocorrelation functions. With the values of the parameters, we can now fit the ARIMA model, and use it to forecast future values of the time series.

In chapter 4, I show the autocorrelation and partial autocorrelation plots, and how to select the values of the parameters of the model (p, d and q) based on empirical data.

[6]Appendix C contains more information about the equations and properties of ARIMA models

Then I fit an ARIMA model using a training set, and test its goodness against a test set. Appendix C provides more details about the equations, parameters and coefficients of an ARIMA model.

CHAPTER
FOUR

Results

4.1 Introduction

This thesis addresses six research questions that were shown in section 1.3.1. I recall here those questions:

1. Are the laws of software evolution valid for the case of libre software?
2. How many metrics do we need to characterize a software product?
3. Which is the shape of the statistical distribution of the size of software?
4. Which are the properties of that distribution? Is there any pattern or commonality in its properties?
5. What kind of dynamics drives software evolution?
6. How can we accurately forecast the evolution of software?

In this chapter, I show the results obtained after applying the methodology shown in chapter 3. The next sections show those results, in the same order that the above list of questions.

4.2 First question: Comparability of different metrics

In section 2.3.4, I highlighted one research question that has not yet been addressed in the literature: the comparability of empirical studies of software evolution that use different metrics. In particular, some works [GT00, RAGBH05] have found that the Linux kernel was evolving at a super-linear rate. The laws of software evolution [Leh85a, Leh96, LRW$^+$97] say that the growth rate of a software system decreases over time because of increasing complexity. Therefore, there is a conflict between those two studies.

SLOC	LOC	FUNCS	CYCLO	HLENG	HVOLU	HLEVE	HMD
0.88	0.89	0.90	0.87	0.87	0.85	0.94	0.77

Table 4.1: *Coefficient of correlation of the number of modules against the rest of metrics. Correlations performed in logarithmic scale.*

The main question behind that conflict is whether or not the laws of software evolution can be applied to the case of libre software.

Lehman labeled the study by Godfrey as an anomaly [LRS01], and said that in any case the studies are not comparable, because the size and time units used by Godfrey are different to those used by Lehman. Godfrey uses SLOC, and Lehman number of files.

4.2.1 Overall correlation

To calculate the overall correlation coefficient at the package level, I took each package as a single point for the correlation analysis. This is, for each one of the $6,556$ packages, I obtained the value for each one the metrics by aggregating the values of the files contained in that package. I decided to calculate the values of the packages by aggregation because the studies with I want to compare to, calculate the values of the overall attributes using the same procedure.

Figure 4.1 shows the scatter plot of SLOC vs Number of modules both in linear (left) and logarithmic scale (right). As that figure suggests, there is a linear trend in the logarithmic plot. Therefore, the correlations will be performed in logarithmic scale. The same behavior is observed when plotting number of modules against the rest of metrics. This is a consequence of the shape of the statistical distribution of software size, which is very close to a normal distribution. This makes it possible to apply the correlation analysis, and to obtain a measure of the significance of the correlation coefficient. All the correlations were computed using the least square methods, and I calculated the Pearson coefficient. I discarded other options, like Spearman's rank correlation, because of the properties of the sample: we have exact measurements of the metrics, and not only ranks.

Therefore, using that sample, I have correlated the number of SLOC, LOC, FUNCS, CYCLO, HLENG, HVOLU, HLEVE and HMD against the number of files in logarithmic scale, for the set of $6,556$ ports written in C. The level of significance for all the correlation coefficients shown in this section is $p < 2.2 \cdot 10^{-16}$ (in other words, all the correlation coefficients are significant). Table 4.1 shows the Pearson coefficients of those correlations. For all the metrics, the correlation coefficient is quite high (only for the case of Halstead's Mental Discriminations, the coefficient is under 0.80). Therefore, at a first glance, all the metrics are highly correlated with number of modules.

The correlations can be biased because of aggregation effects. In order to test the sensitivity of the results with some of the properties of the sample, I divided the overall

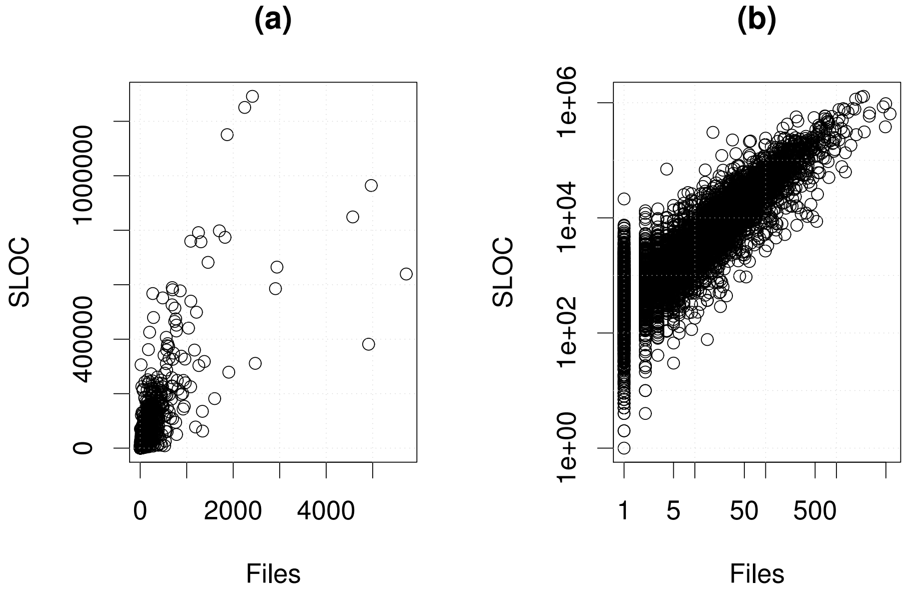

Figure 4.1: Scatter plot of SLOC vs Number of modules. (a) Linear scale. (b) Logarithmic scale. The linear trend is clear in the case of logarithmic scale, but not in the case of linear scale.

sample in smaller and more homogeneous groups. I have divided the groups using the following properties: package size and field of application, which are discussed in the next subsections.

4.2.2 Comparability discriminating by package size

In order to test the sensitivity of the correlations with the size of the packages, I repeated the correlations, but with subsamples obtained by discriminating by package size.

Thus, I divided the distribution of packages in four subsamples. The size limits for each subsample were obtained by calculating the four quantiles of the distribution of the size of the packages. So we ensured that all the subsamples contained the same number of elements.

The correlations were then computed taking all the packages in each subsample, thus obtaining four correlation coefficients. We performed the correlations of number of modules against the rest of metrics. Figure 4.2 shows the results. There is a curve for each correlation, and each curve contains four points, one for each subsample. The size in the horizontal axis is in logarithmic scale, because the difference in the size of the packages is of several orders of magnitude.

As figure 4.2 shows, for small packages, correlations are poor. However, if we take the subsample of the largest packages, correlations are high. Taking into account that the number of points for each correlation is the same (about 1600 points), I cannot assign this poor value to an insufficient number of points in the sample. Therefore, the correlations are only significant for large packages ($> 5,000$ files). For the particular case of SLOC, we can consider that the correlations of this metric with number of modules is

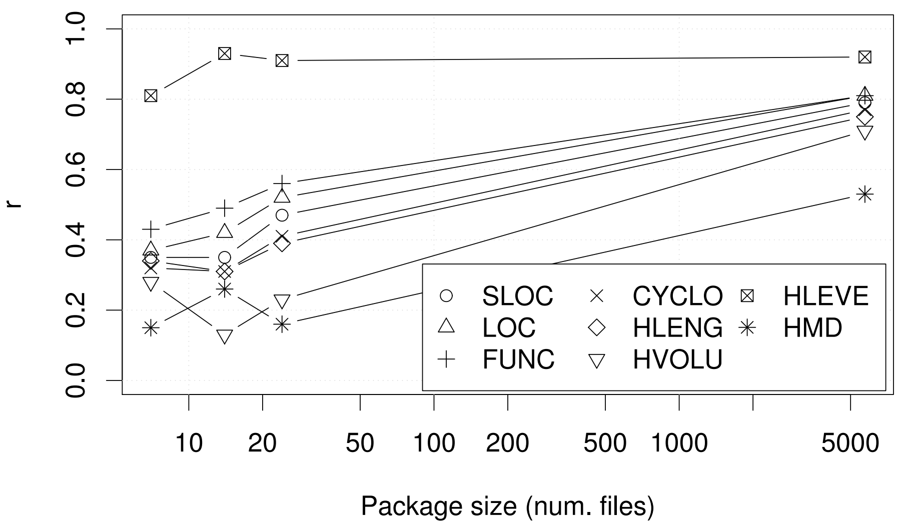

Figure 4.2: *Correlation coefficient against package size. The distribution of packages was divided in four quantiles, and the correlations were performed using all the packages belonging to each interval. This procedure ensured that the number of points for each correlation was constant. The correlations were made in logarithmic scale, of all the metrics against number of modules. For instance, the circles correspond to the correlation of SLOC against number of modules. The correlation is only significant for large packages, probably because of aggregation effects.*

significant for packages containing more than 200 files. There is the exception of the correlation of the number of files with Halstead's level, that is apparently stable.

4.2.3 Comparability discriminating by field of application

In this section, I have divided the whole sample in subsamples, each corresponding to a different field of application. The breakdown process is the same one described in section 3.2.4.

The argument to divide the sample using this criterion is that different fields of application may present different characteristics, and so correlations that are fulfilled in a global sample or in a given field, may not appear in other fields.

I divided the overall sample in 60 subsamples. In each subsample, I calculated the correlation among the number of modules and the rest of size and complexity metrics. The values for each package were obtained by aggregating the values of each individual file. The correlations were performed in logarithmic scale. Therefore, I obtained 60 correlation coefficients for each pair of metrics. Table 4.2 contains the summary statistics for the 60 correlation coefficients. It shows the correlations for each pair of metrics in each row. For instance, the first row shows the summary statistics of the 60 coefficients obtained when correlating SLOC against number of modules in all the subsamples. The last column shows the overall coefficient, obtained using the whole sample (it is the same coefficient shown in table 4.1).

	Min.	Max.	Mean	Median	Sd. dev.	Overall
SLOC	0.78	1.00	**0.88**	**0.89**	0.05	**0.88**
LOC	0.79	1.00	**0.89**	**0.90**	0.04	**0.89**
FUNCS	0.80	1.00	**0.90**	**0.90**	0.04	**0.90**
CYCLO	0.77	1.00	**0.88**	**0.88**	0.05	**0.87**
HLENG	0.79	1.00	**0.88**	**0.88**	0.05	**0.87**
HVOLU	0.77	1.00	**0.94**	**0.94**	0.04	**0.85**
HLEVE	0.76	1.00	**0.86**	**0.86**	0.06	**0.94**
HMD	0.60	1.00	**0.78**	**0.77**	0.08	**0.77**

Table 4.2: *Summary of the correlation coefficients calculated for all the fields of application. All the correlations were made against the number of modules and in logarithmic scale. The last column is the overall coefficient calculated for all the packages as a single sample.*

As that table shows, all the correlations are stable in all the fields of applications. Although there is some variability, the standard deviation is very low, and the mean and the median are very close to the overall coefficient. Therefore these correlations do not depend on the field of application.

4.2.4 Summary

The question addressed here is the comparability of studies that use SLOCs and number of files. Based on the results obtained with a sample of $6,556$ software systems written in C, we conclude that:

> Empirical studies that use SLOC or number of files are comparable, for large software projects ($>$ 200 files), and regardless the field of application of the project.

Applied to the case of the controversy between the studies by Godfrey [GT00] and Lehman [Leh85a], both studies are comparable. Therefore, the laws of software evolution are not valid for the case of Linux (and other cases that use SLOC as size metric instead of number of files).

4.3 Second question: How many metrics to characterize software?

The second research question was about the attributes of software. The two main static attributes of software are size and complexity. But there are many more metrics that can be used to quantify those attributes.

Metric	Mean	Median	Std. dev.	Min.	1st qnt.	3rd qnt.	Max.
SLOC	294	86	829	1	26	284	162,248
LOC	418	144	1,011	1	56	414	162,744
FUNCS	8	2	20	0	0	8	2,253
CYCLO	46	7	123	1	1	40	9,289
HLENG	1,798	449	8,520	2	125	1,576	2,255,154
HVOLU	15,692	2,917	108,234	2	678	11,756	36,116,196
HLEVE	0.0739	0.0284	0.173	~ 0	0.0111	0.0769	2.000
HMD	$3.67 \cdot 10^6$	$1.01 \cdot 10^5$	$3.89 \cdot 10^7$	1	9,508	$1.03 \cdot 10^6$	$2.15 \cdot 10^9$

Table 4.3: *Descriptive statistics of the sample of files. The table shows the mean, standard deviation and quantiles (minimum, first, median, third and maximum) for the different metrics using all the files of the sample.*

Table 4.3 shows some basic statistical properties of the sample used to address this question. The differences between the mean and the median for all the metrics evidence that each one of the variables to be correlated are not normally distributed. The correlation analysis requires all the variables to be normally distributed.

The correlations shown in this section are calculated at the file level, because I think that this is the minimum unit that ensure independence between the items of the sample. In the previous section, I have calculated the correlations aggregating files in order to compare with studies that used the same procedure. However, this aggregation process introduces dependence between the items in the sample. At the file level, if we ensure that files have been independently generated (for example, removing duplicated files), we can affirm that there exist statistical independence between the items of the sample

4.3.1 Correlation for the overall sample

I answer the question of how many metrics (and which ones) using a correlation analysis, over the sample described in section 3.2, and with the metrics described in the same section. The results shown here are bounded to the case of the C programming language, and also to the case of libre software. The correlation analysis was performed at the file level, assuming that all the files have been independently generated. Therefore, for the correlation I had 447,612 points (one for each one of the files), with a set of 8 values for each point (one for each one of the metrics considered). For all the correlation coefficients shown in this section, $p < 2.2 \cdot 10^{-16}$.

The first step is to choose the kind of correlation. I plotted the data in a scatter plot, both in linear and logarithmic scales (see figure 4.3). In linear scale, the relationship is clearly non-linear. In logarithmic scale, although dispersed, the relationship is close to a linear equation. Figure 4.3 shows only Halstead's level versus SLOC, but the same behavior is verified with any other pair of metrics. Because of this, I decided to do all

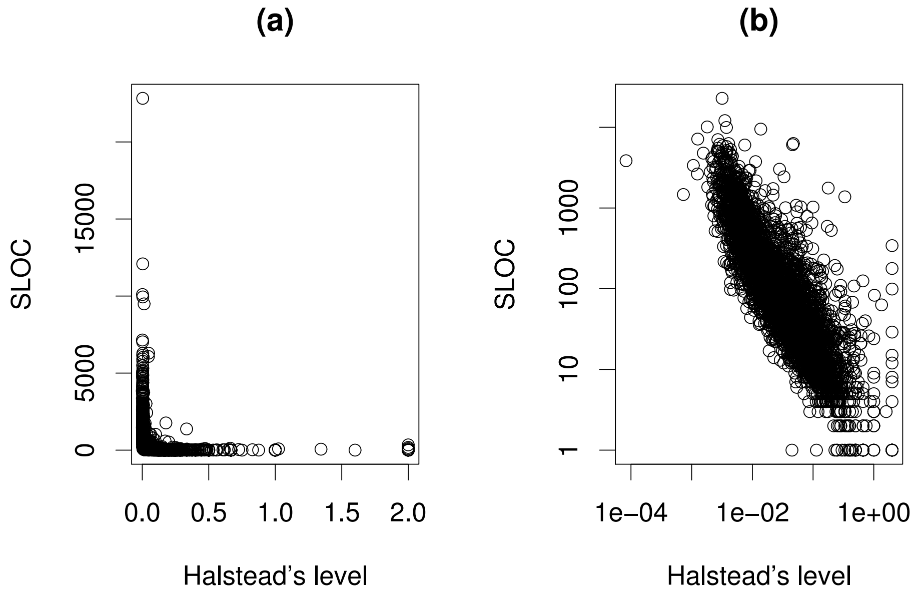

Figure 4.3: *Scatter plot of SLOC against Halstead's level. (a) Linear scale. (b) Logarithmic scale.*

the correlations in logarithmic scale. As in the case of the previous section, I applied the least square method, and I calculated the Pearson coefficient.

Table 4.4 shows the Pearson correlation coefficients among the logarithm of all the metrics. If we focus in the first column, we can see that SLOC is highly correlated with the rest of the metrics. The correlation with the number of functions is not so high ($r < 0.8$). In general, the number of functions presents lower coefficients than the rest of the metrics, except with cyclomatic complexity.

Summarizing, all the complexity metrics are highly correlated with LOC or SLOC. The rest of the metrics are highly correlated as well (the minimum correlation coefficient is 0.72 between the number of functions and Halstead's volume). This means that any of the metrics is providing as much information as any other, and we need only one of them (for instance, SLOC) to characterize the software system.

4.3.2 Correlation discriminating by file type

Table 4.4 shows the correlation for the whole sample of files. All of them are written in C, and the sample contains both header and non-header files. I repeated the analysis using two subsamples, obtained by separating header files ($152,937$ files) from non-header files ($294,675$ files) .

Table 4.5 shows the results. It only shows the correlation coefficients of SLOC against the rest of complexity metrics. The correlation coefficients for non-header files are a little higher than for the whole sample. For header files, they are lower. The difference is quite notable for the case of cyclomatic complexity, which does not show any significant correlation.

	SLOC	LOC	CYCLO	HLENG	HVOLU	HLEVE	HMD	FUNCS
SLOC	**1.00**	0.96	0.82	0.97	0.97	−0.87	0.95	0.76
LOC	**0.96**	1.00	0.79	0.94	0.93	−0.84	0.92	0.74
CYCLO	**0.82**	0.79	1.00	0.80	0.79	−0.86	0.84	0.89
HLENG	**0.97**	0.94	0.80	1.00	1.00	−0.90	0.99	0.73
HVOLU	**0.97**	0.93	0.79	1.00	1.00	−0.90	0.98	0.72
HLEVE	**−0.87**	−0.84	−0.86	−0.90	−0.90	1.00	−0.96	−0.77
HMD	**0.95**	0.92	0.84	0.99	0.98	−0.96	1.00	0.76
FUNCS	**0.76**	0.74	0.89	0.73	0.72	−0.77	0.76	1.00

Table 4.4: *Correlation among the logarithm of all the metrics. This sample includes both header and non-header files.*

	CYCLO	HLENG	HVOLU	HLEVE	HMD
Header	**0.35**	0.92	0.92	−0.71	0.89
Non-header	0.91	0.98	0.97	−0.90	0.96

Table 4.5: *Correlation coefficients of SLOC against the rest of complexity metrics for the header and non-header files.*

This difference is obviously due to the nature of header files. They usually lack loop structures, and branching of the flow of the program. Header files are flat, containing declarations of variables, functions, etc. Therefore, regardless the size of the file, the cyclomatic complexity value is low.

Summarizing, the main conclusion is that for header files the high correlations between size and cyclomatic complexity are not verified. Halstead's science metrics are highly correlated both for header and non-header files.

4.3.3 Correlation discriminating by package size

In this section, I explore how the correlation coefficient among SLOC and the set of complexity metrics varies with the size of the package. I performed the correlations in two different ways:

- Across packages
  I divided the set of packages in nine intervals, being the frontier between intervals the values of the deciles of the sample of the sizes of the packages. Then I took all the files belonging to the packages in each interval and I calculated the correlation coefficients using those files.

- Within each package
  I computed the correlation coefficient using the sample of files contained in each package, and then I plotted the correlation coefficient for each package against its

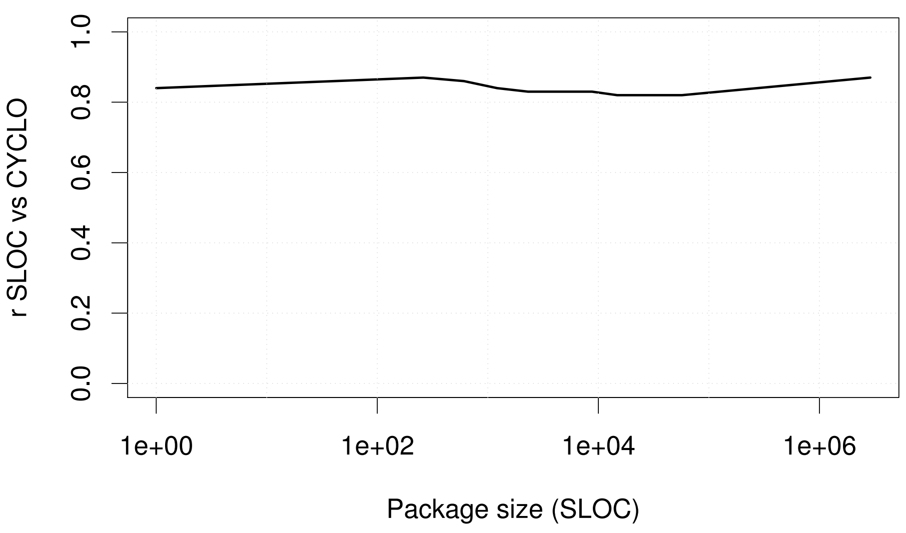

Figure 4.4: *Correlation coefficient (SLOC vs McCabe, logarithmic correlation) against package size. The size of the packages was divided in nine intervals (deciles), and the correlation coefficient was calculated using all the files belonging to the packages contained in each decile (this is, across packages). Horizontal axis in logarithmic scale.*

size. I obtained a very dispersed scatter plot, so I summarized the data using a smoother.

Correlation across packages

As shown in the previous chapter, the files contained in the sample under study belonged to $6,556$ different packages. I measured the size of each one of these packages, then I calculated the deciles for this sample. This process split the sample in nine intervals. For all the packages in each interval, I took all the files contained in those packages as a sample, and calculated the correlation coefficients in logarithmic scale. I calculated the correlations for SLOC against the rest of complexity metrics.

Figure 4.4 shows the results for the correlations across packages, for the case of the SLOC against cyclomatic complexity. The correlation coefficient is quite stable when the size varies. The rest of correlation coefficients shows the same behavior. Therefore the size of the package does not affect the strength of the correlation. The high correlation coefficients obtained with the whole sample are not because of aggregation effects (at least, of aggregation at a global level, because we are aggregating packages belonging to the same interval).

Correlation within packages

In this case I took all files in each package, and I used them as sample to calculate the correlations. Therefore, we had a correlation coefficient for each package. In this case, of course, the results are much more scattered. Figure 4.5 shows the scatter plot of

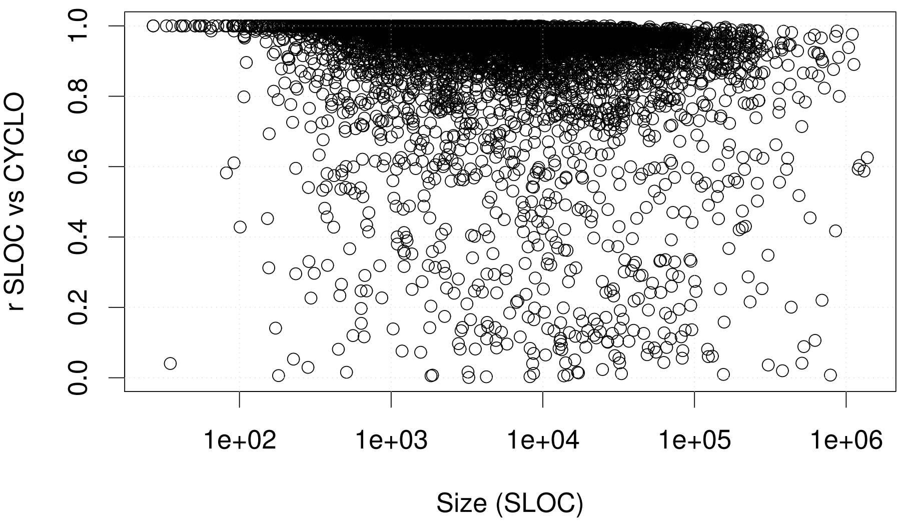

Figure 4.5: *Scatter plot of the correlation coefficient of SLOC vs CYCLO for each package, against the size of the package in SLOC. Each point represents a package. We have then* 6,556 *points in this plot. The correlation was logarithmic. Although the results are very dispersed, we can observe that most of the points concentrate on values of around* $r = 0.80$. *Horizontal axis in logarithmic scale.*

the correlation coefficient of SLOC against cyclomatic complexity. The correlation was logarithmic. In this case, there is great variation in the value of the correlation coefficient for each package, but we can observe a main trend in the graph.

In order to extract the trend of the data shown in figure 4.5, I used the *lowess* smoother [Cle81]. Figure 4.6 shows the results. This time, the trend is much clearer. The results are similar to those shown in figure 4.4. However, in the case of correlations within packages, the correlation coefficient falls for large packages. For the case of small packages the results are not very significant, because those packages contain very few files to be enough to perform a correlation using them. In any case, the correlation coefficient is not heavily affected by the size of the package. I can conclude then again that the strength of the correlation of SLOC and CYCLO using the whole sample is not because of any aggregation effect.

I repeated the same procedure for the rest of the correlations of SLOC against the rest of complexity metrics, in logarithmic scale. All the metrics but Halstead's level showed also stable. In the case of Halstead's level, for large packages the correlation is poor. That could indicate that the high correlation coefficients obtained when using the whole sample are because of aggregation effects. In other words, I cannot affirm that there is indeed a correlation among SLOC and Halstead's level. Figure 4.7 shows the results for these metrics. Again, the figure shows only the main trend obtained using *lowess*.

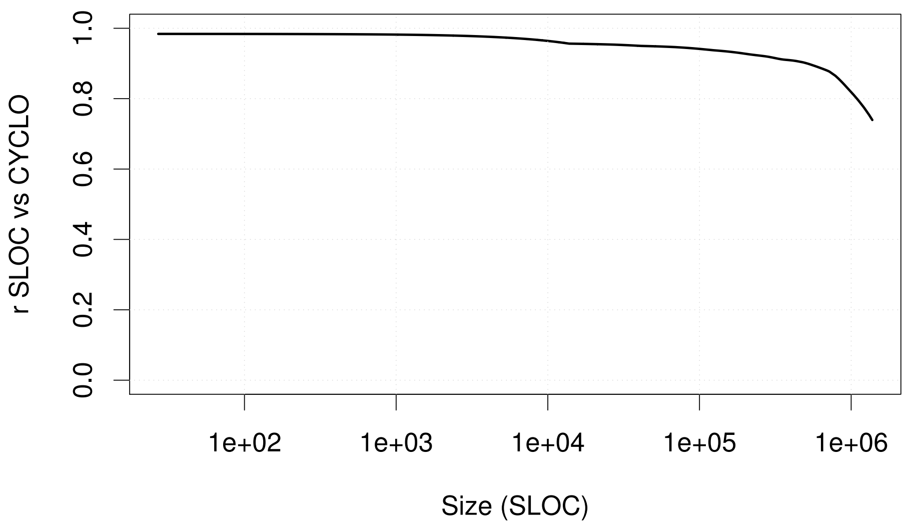

Figure 4.6: *Smoothing of the correlation coefficient of SLOC vs CYCLO for each package, against the size of the package in SLOC. The correlation was logarithmic. Horizontal axis in logarithmic scale.*

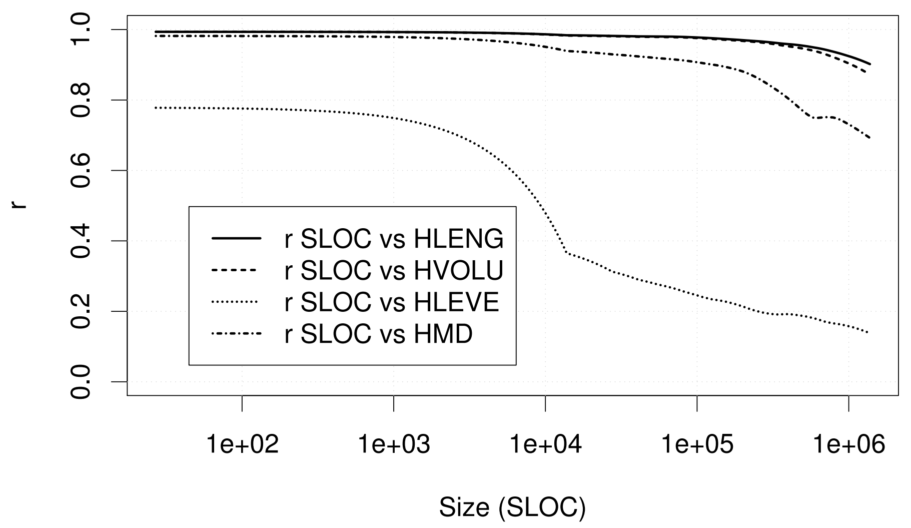

Figure 4.7: *Smoothing of the correlation coefficients of SLOC vs all the Halstead's metrics for each package, against the size of the package in SLOC. The correlation was logarithmic. Horizontal axis in logarithmic scale. It seems that the correlation of SLOC vs Halstead's level is not stable with the size of the package. Therefore in spite of the high coefficient when correlated globally, we cannot conclude that there is indeed a correlation between the two variables.*

	Min	Max	Mean	Median	Sd. dev.	Overall
CYCLO	0.66	0.99	**0.84**	**0.85**	0.05	**0.82**
HLENG	0.88	1.00	**0.97**	**0.97**	0.02	**0.97**
HVOLU	0.86	1.00	**0.97**	**0.97**	0.02	**0.97**
HLEVE	−0.99	−0.38	**−0.57**	**−0.56**	0.10	**−0.87**
HMD	0.87	1.00	**0.95**	**0.96**	0.02	**0.95**

Table 4.6: *Summary of correlation coefficients calculated by field of application. Correlations against SLOC. The last column shows the correlation coefficient computed for the whole sample, without any classification (by filetype, by package size, etc).*

4.3.4 Correlation discriminating by field of application

Different field of applications have a different programming culture, and they address problems with different levels of complexity. For example, the code of an operating system may not show the same characteristics that an end-user application such as web browser. Because of that, the results obtained in the previous sections may greatly vary depending on the field of application.

The overall sample in this case was obtained from software packages classified in 60 different categories. Examples of these categories are `net`, `devel`, `games`, `www`, `sysutils`, `databases`, etc. Therefore, it is a rich sample of many different field of applications.

In order to measure the sensitivity of the results to the field of application, I repeated the correlations for the files contained in each one of the categories. So I obtained 60 correlation coefficients for each pair of metrics. I computed the correlations among the logarithm of the metrics. The results are summarized in table 4.6. All the correlations in that table are made against SLOC. For instance, the first row contains the linear correlation coefficients of SLOC vs CYCLO. Because there are 60 values, that table shows only a summary of the main statistics of this set of 60 coefficients.

As that table shows, the behavior is quite stable for all the pairs of metrics, except for SLOC vs Halstead's level (I discuss this case below). For instance, the overall coefficient shown in table 4.4 is quite similar to the mean and median values of the coefficients obtained by discriminating by field of application.

The case of SLOC vs Halstead's level (HLEVE row in table 4.6) deserves more attention. In the previous section, I showed that the logarithmic correlation of SLOC vs Halstead's level was sensitive to the size of the package. When the package size reaches certain value, the correlation coefficient drops, indicating no relationship among the two metrics. The coefficient presents a mean value of -0.57 and a median of -0.56, in spite of the overall coefficient of -0.87 and a maximum value of -0.99 (and some other values of ~ -0.80).

The correlation coefficient of SLOC vs Halstead's level is quite sensible to the field of application. This would indicate that, as it happened in the previous section, the good overall coefficient is due to aggregation effects. Therefore I cannot affirm then that there is indeed a correlation among SLOC and Halstead's level.

4.3.5 Analysis of Halstead's level

Let me analyze this exception in more detail. Halstead's level definition was shown in equation 3.4, in section 3.2.3. I reproduce here that equation:

$$lv = \frac{2}{n_1}\frac{n_2}{N_2} \tag{4.1}$$

In that equation n_1 is the number of distinct operands, n_2 is the number of distinct operators, and N_2 is the total number of operators. The definition of operand and operators was given in the mentioned section[1]. Higher Halstead's levels imply less difficult programs. Actually, the inverse of level is denominated sometimes as *difficulty*. The $\frac{n_2}{N_2}$ element measures the inverse of the average number of operators used in the code (compared against the number of different operators used). Using the above definition, the more operators are used in the code, the more difficult is to comprehend the program. In other words, the difficulty of the program is directly proportional to the redundancy of statements.

The other element $\frac{2}{n_1}$ measures the inverse of the number of operands. The more times operands are used (regardless they are defined as new or reused as instantiations), the more difficult the program will be.

I think that the average number of operators in a source code file must be probably independent of the size, and it has to be more related for instance to the programming language.

I have not measured other data that could help to relate the value of Halstead's level to the actual complexity of the file (like number of hours spent while writing the file, or number of defects that the file contained).

4.3.6 Summary

In this section, the second research question has been discussed: how many metrics do we have to use to characterize software? For answering it, I considered the two main static attributes of software: size and complexity, measured using eight different metrics.

For an overall sample of libre software written in C, all the metrics are highly correlated. Therefore, in principle, from that set of eight metrics, we only need one to characterize software. However, when correlating Halstead's level against the rest of metrics, using more homogeneous samples (filtering the overall sample by package size or by field of application), I found many cases where the correlation was not verified. Therefore, we can use SLOC and Halstead's level to characterize a software system.

For header files, the correlation of SLOC vs McCabe's cyclomatic complexity is very poor for header files. Therefore, we should use McCabe's cyclomatic complexity as well as any of the rest of the metrics.

[1]In a nutshell, operators are statements, and operands are variable definitions and instantiations.

4.4 Third question: The shape of the distribution of the size of software

The third question is about the shape of the statistical distribution of the size of software. The size of the sample used for the previous analysis allows us to obtain an estimation of the statistical properties of software. The shape of the distribution is important, because it helps to determine which parameters of the distribution better describe it. For instance, if the distribution is normal, we can use the mean and the standard deviation. Furthermore, the shape of the distribution provides information about the process that generated that distribution.

Table 4.3 on page 70 shows the main statistical properties (mean, median, etc) for the case of the sample of software written in C. The difference between the median and the mean, indicates that the density function is right skewed. To explore this issue, I show here the shape of the distribution of the SLOC metric, and considering the complete sample of $12,010$ packages (this includes packages written in different programming languages). I have tested that the shape of the distribution does not change considering only packages written in C, and all the software packages.

After plotting an histogram of the sample and an estimation of the probability density function, I could not find a particular statistical distribution for this result. However, after repeating the procedure of estimating the density function using the logarithm of the values of the metrics, all the cases were apparently close to a normal distribution. I decided to explore further to try to find out whether or not the sample matched with a particular statistical distribution.

Figure 4.8 shows a normality test for the logarithm of the SLOC metric. That test compares the quantile of the sample with the quantiles of a theoretical normal distribution. If the sample were extracted from a normal distribution, the scatter plot of the quantiles would be a straight line. In the case of figure 4.8, it shows that only in a certain range the sample follows a straight line. In particular, tails (lower and upper) deviate from normality.

I could not figure out the shape of the distribution for the tails using only the quantile-quantile plot. In order to find out the profile of the tails, we can use the complementary cumulative distribution function (CCDF) plot. Figure 4.8 shows that the tails start to deviate from normality for values higher than $3,000$ SLOC ($\sim 7-8$ in logarithmic scale), and lower than 10 SLOC (~ 2 in logarithmic scale). Figure 4.9 shows the plot CCDF of the tails (left side for the lower tail, right side for the upper tail).

The upper tail follows a power law distribution, because its complementary cumulative distribution function is close to a straight line. The lower tail seems not to be a power law distribution (neither a normal distribution as shown in figure 4.8).

If we look at the whole plot of the complementary distribution function, we can observe this combination of a lognormal body with a power law tail. Figure 4.10 shows that plot. It shows the CCDF of the sample of SLOC, a lognormal density function with the same mean and standard deviation than our sample, and a power law density

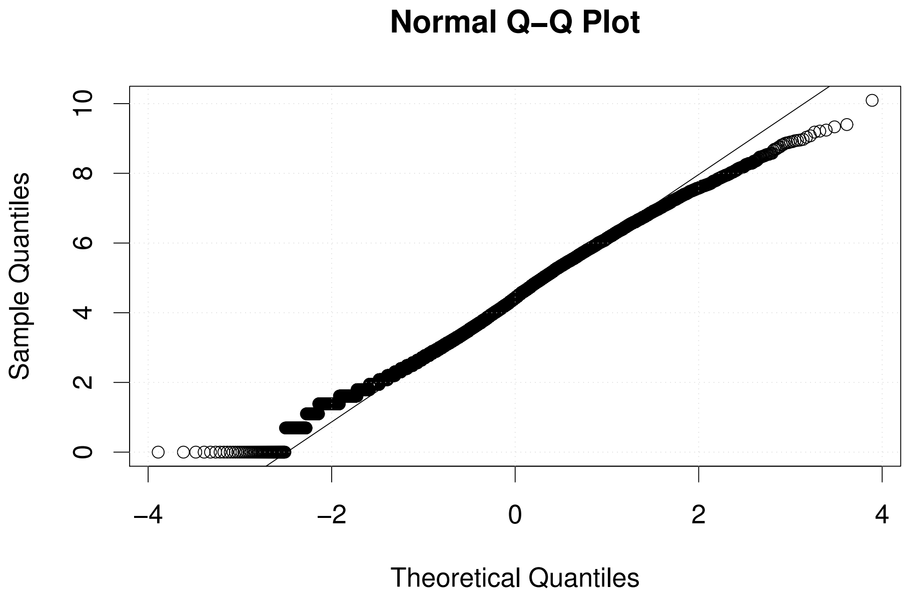

Figure 4.8: *Normality test for the final sample (SLOC metric). Tails deviate from normality, but the main body of the sample is very close to a normal distribution. The test is done in logarithmic scale.*

function estimated for the upper tail (SLOC$>$ 3000) using the methods proposed by Clauset *et al.* [CSN07]. The main body matches well the lognormal, the tail of the sample deviate from the lognormal and it is well fitted by a power law[2].

That profile (lognormal body, power law tail) has been identified before [Mit04b]. It is called a *double Pareto distribution*. In that distribution, the tails for low or high values follow power law distributions, while the body follows a lognormal distribution[3].

4.4.1 Summary

This third question is about which is the shape of the statistical distribution of the size of software. In the previous sections I found that all the metrics were highly correlated when using the overall sample. For that same sample, I showed the main statistical properties. Those properties are only estimators of the actual properties of the whole population.

But more important than the actual values of the mean, the median or any other property, is the shape and parameters of the distribution. Software size and complexity is distributed like a double Pareto distribution, which has a lognormal body and power law tails. In this particular case, the power law tails do not have a heavy influence. Therefore, we can approximate the population by a lognormal, and use the mean and standard deviation of the lognormal as estimators of the whole population. This allows

[2]Appendix B includes plots and tables for the rest of metrics, and for header and non-header files.

[3]Appendix A shows a description of this distribution and some of its properties.

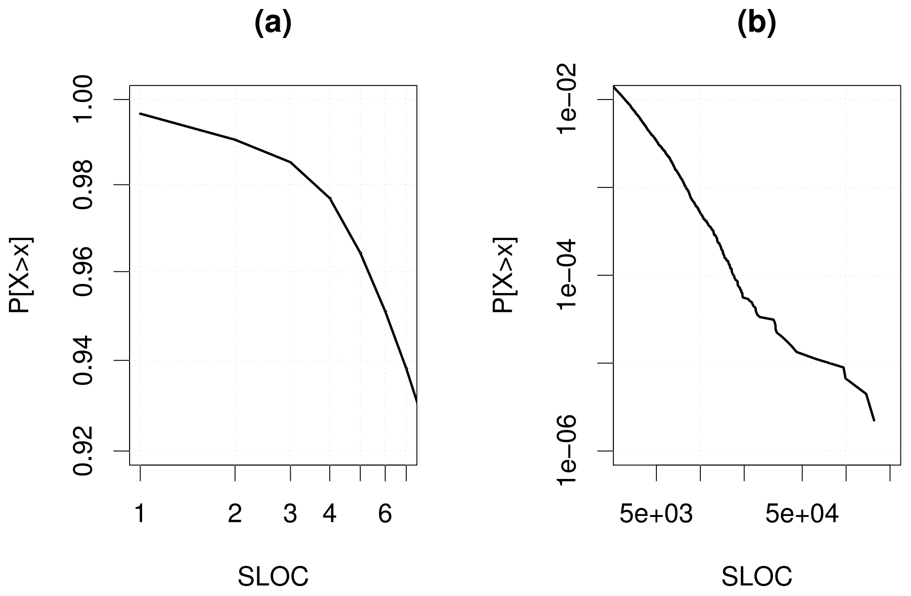

Figure 4.9: *Complementary cumulative distribution function for the SLOC sample. (a) Lower tail (SLOC* < 10*). (b) Upper tail (SLOC* > 3000*). The upper tail is a straight line, indicating that the upper values follow a power law distribution.*

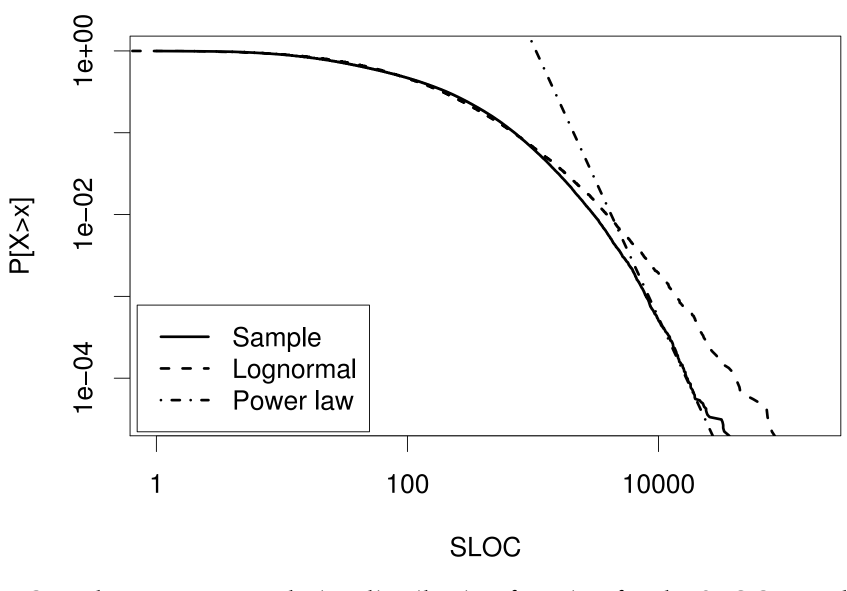

Figure 4.10: *Complementary cumulative distribution function for the SLOC sample, compared against a lognormal distribution with the same mean and standard deviation, and against a power law distribution with* $\alpha = 3.34$*. The high tail of the sample deviate from the lognormal, and it is close to a power law distribution (straight line).*

for some affirmations. For instance, for the case of files written in C, table 4.3 shows that half of the files has less than 144 lines of code (86 SLOC). Only 25% of the files have more than 414 lines of code (284 SLOC).

4.5 Fourth question: Self-similarity in software

The fourth question is about patterns in the attributes of software. In the previous section I have found that software size is distributed like a double Pareto. In this section, I show whether this distribution appears using other metrics, for the whole sample and also for more coherent subsamples (using the same breakdown criteria shown in the previous sections).

4.5.1 Header and non-header files

In section 4.3, I found that the cyclomatic complexity of header files was not highly correlated with size (unlike the rest of complexity metrics). Because of that, I looked at the statistical distribution of all the metrics of both header and non-header files.

Considering the SLOC metric, when plotting the quantile-quantile normality plot, and the complementary cumulative distribution function, the shapes of the plots are the same that those shown in figures 4.8 and 4.9, both for the case of header and non-header files.

Figure 4.11 shows the quantile-quantile normality test for the logarithm of the cyclomatic complexity of the sample of header files. Again, we have the same behavior that with SLOC, but this time the low end tail seems to be more heavy; that is because a lot of header files have very low values for complexity (1 in linear scale, 0 in logarithmic scale). Figure 4.12 shows the complementary cumulative distribution function (CCDF) of the sample, compared against a lognormal with the same mean and standard deviation that the lognormal part of the sample, and a power law distribution fitted using the procedure proposed by Clauset *et al*. [CSN07]. As that plot shows, the CCDF does not match a lognormal. A pure power law distribution would be a straight line in that plot, but the sample is clearly not a straight line. Therefore, the CCDF of the sample falls between the extreme cases of power law and lognormal. However, this time, the power law character is stronger than in the rest of cases. In the case of non-header files, the behavior is exactly the same than with the overall sample.

Therefore, the distribution of size both for header and non-header files is similar than in the case of the overall sample. However, the distribution of complexity for header files seems to have a different shape, much closer to a power law distribution than in the case of non-header files.

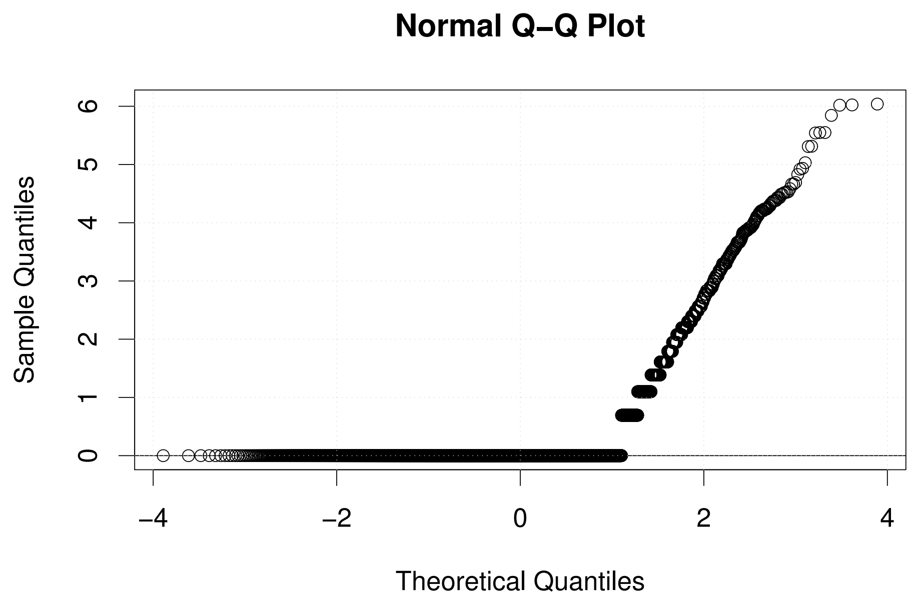

Figure 4.11: Normality test for the logarithm cyclomatic complexity of header files. Low tail deviate from normality and the main body is lognormal as in the case of SLOC. Very high values also deviate from normality.

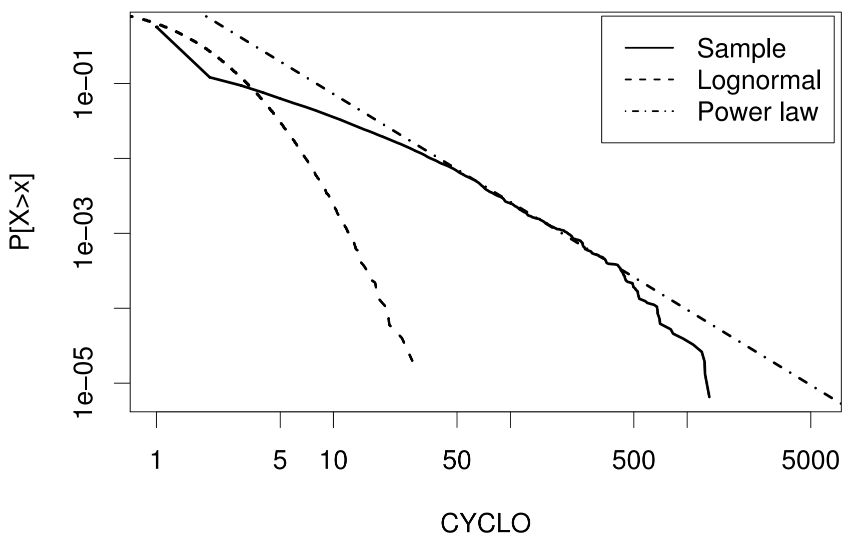

Figure 4.12: Complementary cumulative distribution function for the cyclomatic complexity of header files, and comparison against a lognormal with the same mean and standard deviation, and against a power law distribution with $\alpha = 2.44$. The sample is not fitted by a power law neither by a lognormal.

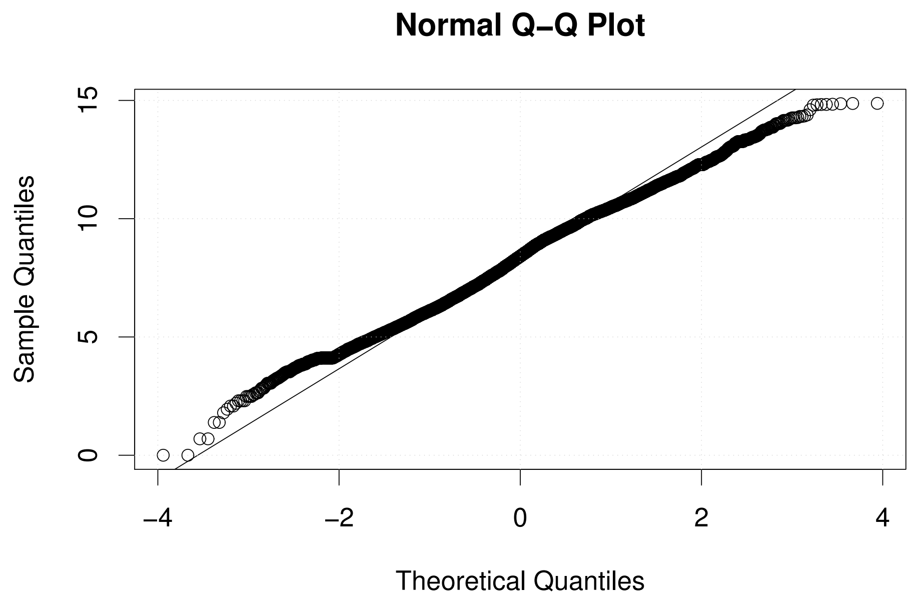

Figure 4.13: *Normality test of the logarithm of package size. The main body of the distribution is normal, and the tails deviate from normality.*

4.5.2 Shape at more aggregated levels

As shown in table 3.1 on page 47, the sample had $12,010$ unique software packages, this is, packages that were not different versions of the same package. Although in all the previous analysis included in this section I have focused in the case of the C programming language, we measured the number of SLOC for all the files in all the packages. Hence, we can represent the distribution of size using all of them.

Figure 4.13 contains a normality test of the logarithm of the size of software packages. As in the previous cases, the sample is very close to a lognormal distribution, although the tails of the sample seem to deviate from the lognormal.

In order to find out the distribution that matches the shape of the tails, we can use the complementary cumulative distribution function. Figure 4.14 shows it compared against the CCDF of a lognormal distribution of the same mean and standard deviation, and a power law distribution estimated using the method proposed by Clauset *et al.* [CSN07].

If we aggregate one more level, and represent the size of sections (as aggregated number of SLOC), we obtain the same kind of distribution. Figure 4.15 shows the CCDF of the size of sections, compared against the CCDF of a lognormal with the same mean and standard deviation, and a power law distribution estimated using the methods proposed by Clauset *et al.* [CSN07]. The figure shows again how much the tail deviates from the lognormal, indicating the same shape than in all the previous cases.

The same kind of distributions appears if I use number of elements instead of number of SLOC. For ports, the elements that compound them are files, and for sections the elements are ports. If I repeat the analysis using the number of files to measure the size

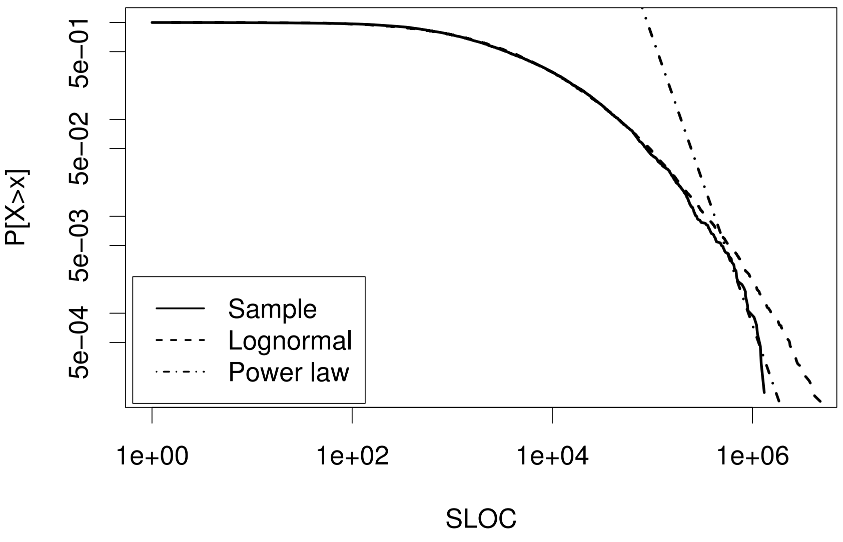

Figure 4.14: *Complementary cumulative distribution function of the size of the ports (software packages). The empirical CCDF is compared against a lognormal distribution, with the same mean and standard deviation, and against a power law distribution with* $\alpha = 2.97$*. The body of the sample matches the lognormal distribution, but the tail is better described by a power law.*

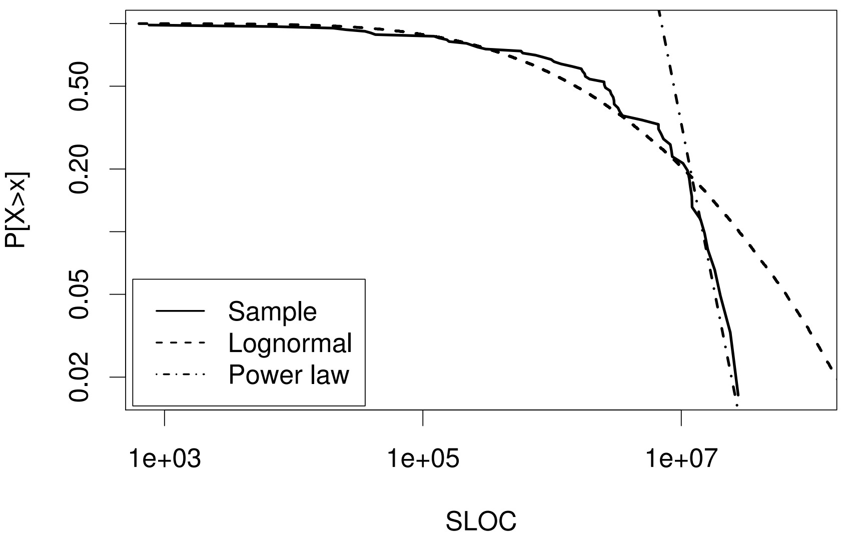

Figure 4.15: *Complementary cumulative distribution function of the size of the sections (or fields of applications). The empirical CCDF is compared against a lognormal distribution, with the same mean and standard deviation, and against a power law distribution with* $\alpha = 3.14$*. The body of the sample matches the lognormal distribution, but the tail is better described by a power law.*

of packages, and the number of packages to measure the number of sections, we obtain the same type of distribution. Similarly, if I consider only those packages that contain files written in C, the same results are obtained.

Summarizing, the same distribution of size is found at different levels of granularity. If I measure the size of the sections, I obtain a double Pareto distribution. If we zoom in, and measure the size of ports, the same distribution appear. Zooming in again, I obtain once more the same distribution for the size of files.

4.5.3 Summary

This section has shown how the double Pareto distribution appears at other levels of granularity, and that the shape of the distribution of size of header files is different, much closer to a power law distribution.

The previous section showed how that distribution appears when measuring the size of files. In this section, I have shown that if I measure the size of packages, either using SLOC or number of files within each package, that parameter is distributed like a double Pareto. If I measure the number of domains of application, or sections in the *ports* argot, either using SLOC, number of files or number of packages within each section, that parameter is distributed like a double Pareto as well.

This means that the same shape is observed at different levels of granularity. In other words, software is self-similar in this respect.

4.6 Fifth question: The dynamics of software evolution

In this section I present the analysis of the dynamics of software evolution, showing how to apply time series analysis to analyze software evolution. The main goal is to answer the question of whether evolution is a short memory or a long memory process.

For this analysis, I have used a sample of $3,821$ projects. The properties of the projects and the method used to retrieve the data were described in section 3.3. For each project I had a time series of the daily number of commits, and another time series of the daily number of modification requests (MRs).

For each one of the projects, I computed the autocorrelation coefficients function (ACF) of each time series, obtaining $3,821$ profile plots like the shown in section 3.3.2.

I correlated the autocorrelation coefficients against the time lag, obtaining the Pearson coefficient for each project. The Pearson coefficient measures the linear correlation between two variables, with a value within the interval $[-1,1]$. The closer its absolute value is to 1, the stronger the linear relationship is. In other words, if two variables do not have a linear relationship, the absolute value of the Pearson coefficient would be much lower than 1.

If the process is close to an ideal short term process, the Pearson coefficient should be close to 1. On the other hand, if the process is not short term, the coefficient should indicate no linear relationship among the parameters.

	Minimum	Maximum	Mean	Median	Sd. dev.
Commits	0.3235	0.9998	0.8429	0.8534	0.1238
MRs	0.2886	0.9998	0.8268	0.8374	0.1214

Table 4.7: *Statistical properties of the set of Pearson correlation coefficients. The values range from very low (~ 0.3, corresponding to long term processes) to very high (~ 0.99, corresponding to short term processes). With only this information, we cannot quantify how many projects could be classified in each category (short or long term).*

I repeated the mentioned procedure for the 3,821 projects. The result was two sets 3,821 Pearson coefficients (one for number of commits, and another one for number of modification requests). As shown in table 4.7, the values of the coefficients ranged from ~ 0.3 to ~ 0.99. For all the correlation coefficients, $2.2 \cdot 10^{-16} < p < 0.38$ (the highest p values correspond to the lowest Pearson coefficients).

At a first glance, it seems that the values are distributed around high values (this is for instance $r > 0.8$), which would indicate that the processes under study are closer to a short term process than to a long term one.

To quantify how many projects could be classified as short term or long term, I estimated the probability density function of the distribution of coefficients, plotted the boxplot and calculated the quantiles of the sample.

Figure 4.16 shows the boxplot, for both number of commits and number of MRs. The results are quite clear. Focusing on the low values, the boxplot indicates that the values below 0.5 (approximately) may be considered outliers. Most of the values are in the range between 0.75 and 0.95, and the median is around 0.85. Summarizing, most of the projects present a high Pearson coefficient, suggesting a strong linear relationship (both for the case of number of commits, and number of MRs). This would indicate that most of the projects are evolving like short range correlated processes.

Figure 4.17 shows the estimation of the probability density function. There is a group of projects with very high values, and another group around a value of approximately 0.85 (this is, a group around the value of the mean or the median). Regarding the low end of the distribution (this would represent long term correlated projects), those projects are only a minority. This behavior is verified both with the number of commits, and with the number of MRs.

A more accurate statistical tool to quantify the number of projects is the estimation of the quantiles of the sample. Table 4.8 contains the quantiles for the Pearson coefficients. For instance, only 40% of the projects present a correlation coefficient lower than 0.80 (0.8178 for number of commits, 0.8036 for number of MRs), and only 20% lower than 0.7394 for number of changes, or 0.7248 for number of MRs. In other words, 80% of the projects have a coefficient greater than ~ 0.72.

According to the results shown in that table, and taking into account the statistical analysis performed on the Pearson coefficients, most of the projects are governed by a short memory dynamics. There are only a few projects that present a profile that would

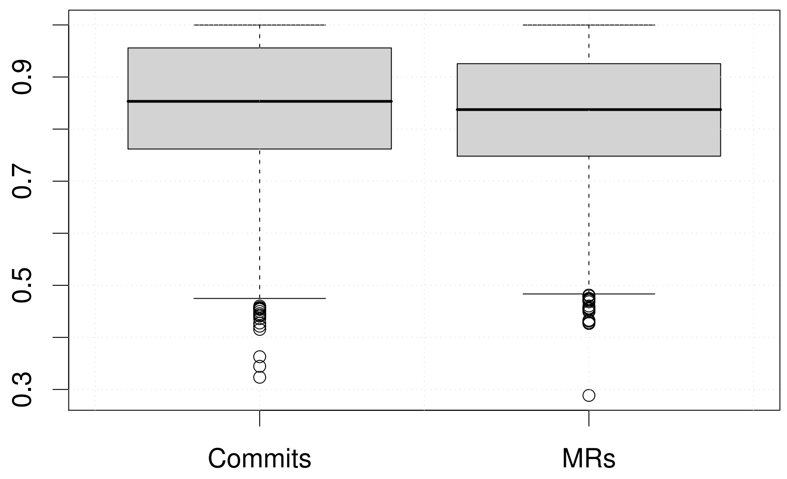

Figure 4.16: *Boxplot of the set of Pearson correlation coefficients, for number of commits, and for number of MRs. This graphs shows that most of the cases (surrounded by the main box in the plot) are around high values (∼ 0.85) of the coefficient. In other words, most of the projects appear to be short term processes.*

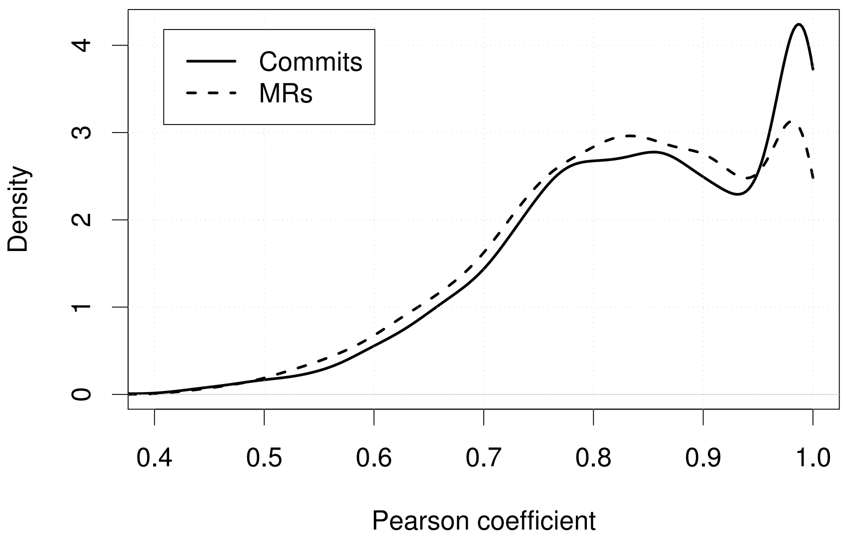

Figure 4.17: *Estimation of the probability density function of the set of Pearson correlation coefficients for number of commits and number of MRs. There is a group of projects with coefficients very close to 1, and another group symmetrically distributed around a value of approximately 0.85. Both groups would correspond to short term processes. Long term processes, located in the left tail, are a minority.*

	0	**20**	**40**	**60**	**80**	**100**
Commits	0.3235	0.7394	0.8178	0.8906	0.9783	0.9998
MRs	0.2886	0.7248	0.8036	0.8705	0.9464	0.9998

Table 4.8: *Quantiles (in %) of the sample of Pearson correlation coefficients, for number of commits, and number of MRs. Less than* 40% *of the projects present a value lower than* $\sim$ 0.8.

indicate a long memory dynamics. This behavior is verified both measuring activity as the number of commits, and as the number of MRs.

4.6.1 Sensitivity analysis

The projects in the sample under study are very heterogeneous. For instance, the size of the project varies in a wide range, as well as the number of developers, age, or any other parameter. It could happen for instance that the number of developers or the size of the project influences how it evolves.

In order to find out if the results are sensitive to some of the properties of the project, I performed a brief sensitivity analysis.

I considered all the factors that are shown in table 3.3 on page 56. I plotted the values of each one of those properties against the value of the Pearson coefficient calculated in the previous section. Thus, I can determine if there are patterns when the projects are clustered in homogeneous groups (for instance, I could find that small projects evolve like short term processes, but large projects do not).

Figure 4.18 shows the results of the sensitivity analysis, only for the case of number of commits (the plots with the Pearson coefficients of number of MRs are very similar). That figure contains six plots. Each plot compares the value of one of the properties shown in table 3.3 against the value of the Pearson correlation coefficient (for the case of number of commits). Each point corresponds to a project. Some of the plots have their vertical axis in logarithmic scale. This is because the kind of distribution of that property. For instance, size is distributed following a Pareto-like distribution (there are a few projects that are very large compared to the rest). That kind of data does not show well in linear scale, as only some isolated points appear in a side of the plot, and a set of points grouped in a small area in the other. In order to make the plots clearer, I selected the logarithmic vertical axis for those properties. The horizontal axis is in linear scale, and shows the Pearson correlation coefficient. Values of the coefficient close to 1 correspond to short term processes. Values lower than 0.7 may be considered long term processes.

At a first glance, there is no any clear pattern in the data. In other words, in spite of the heterogeneity of the projects, the character of the process (short or long term) is not related to any of the properties.

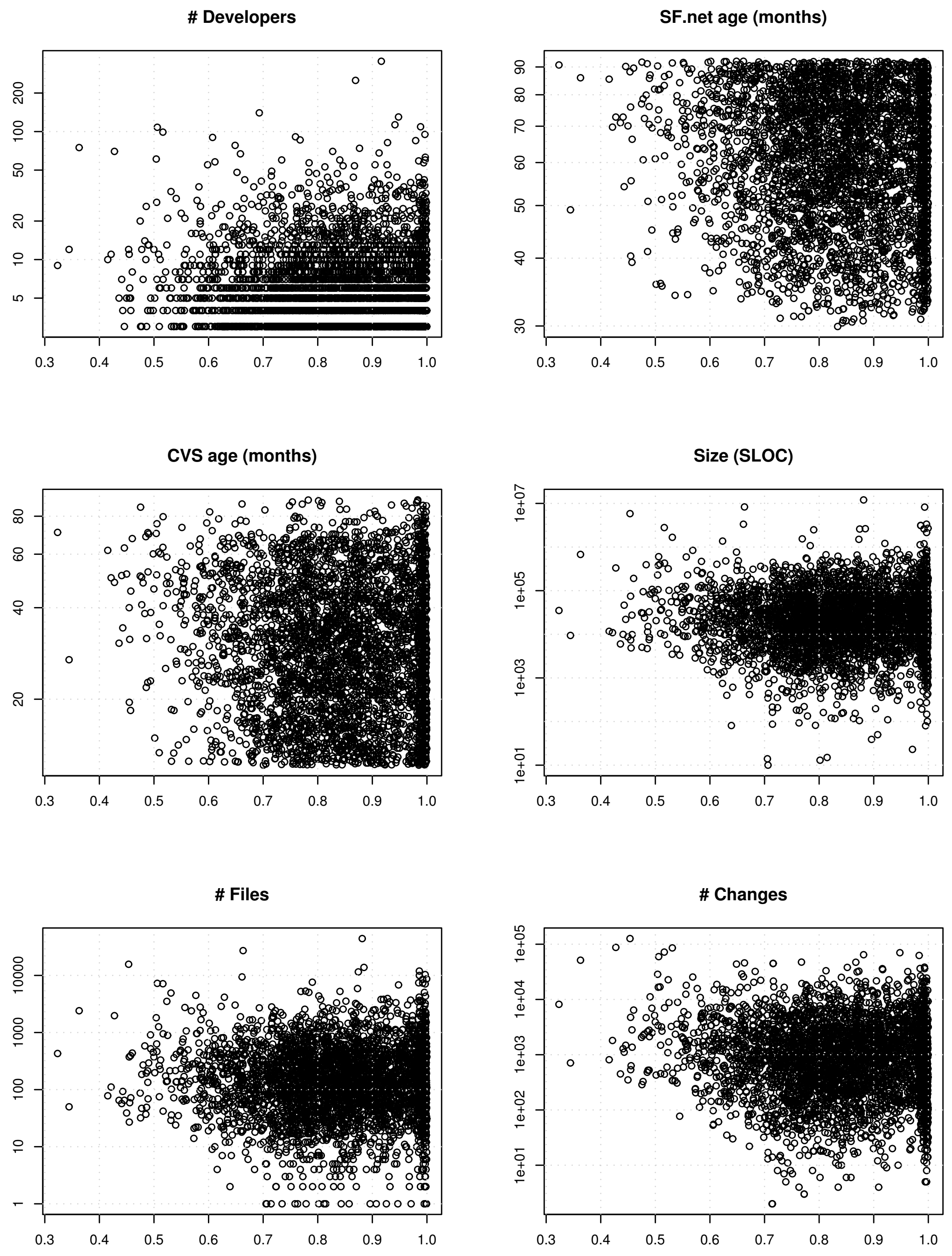

Figure 4.18: *Sensitivity analysis. The plots show the values of each one of the properties shown of the projects (vertical axis), compared against the Pearson correlation coefficient for the number of changes series (horizontal axis). Short term projects present values close to* 1*. The vertical axis of some of the plots are in logarithmic scale. The plots show that there is no pattern when dividing the projects in more homogeneous groups. For all the ranges of the Pearson coefficients, we find projects with values of the properties in a wide range. In the same way, for all the ranges in each one of the properties, we can find projects with a wide range of values of the Pearson coefficients.*

For instance, focusing in the case of size, in the range of the Pearson coefficient from 0.9 to 1.0, there are small, medium and large projects. If I focus in a range of size, I also find projects with a wide range of values of the Pearson coefficient.

This behavior is verified with the rest of properties too. However, the amount of dispersion is different for each property. For instance, the dispersion of the points for the case of CVS age is larger than the dispersion for the case of size. This is probably because the statistical distributions of those two properties are different. In other words, in the case of size, there are only a few points for very large and very small projects in the plots, but that is because indeed there are only a few projects with those values in the sample.

4.6.2 Summary

After analyzing the shape of the profiles of the time series, the main result is that at least 80% of the projects can be considered short term processes. There are two different groups of short memory projects, that are statistically different. However, I could not identify which were the members of each group. Regarding those projects evolving like long term processes, I have not found a group with that characteristic. The long term evolving projects are a marginality compared to the population of the rest of the projects.

The results were not sensitive to any other properties of the projects (like age, size, number of developers, etc). Therefore, although the projects under study do not form a homogeneous group, the short term evolution dynamics is present in projects with very different properties, suggesting that this dynamics may be an universal property.

4.7 Sixth question: Forecasting software evolution

In this section, I present the results of fitting and modelling evolution using ARIMA models. I show how to obtain the parameters the ARIMA models to forecast the near future of a software system, by applying the Box-Jenkins method. ARIMA models are accurate to predict short memory processes, with linear profiles of the ACF plot. As shown in the previous section, software evolution is of that kind. Therefore, ARIMA models are a good option to obtain a statistical model of software evolution.

As it was shown in section 3.4, for the modelling analysis we used three case studies: FreeBSD, NetBSD and PostgreSQL. I measured the size in SLOC for all of them, using the CVS repositories.

Figure 4.19 shows the growth curves for all the case studies. Those curves are very close to linear. When correlating SLOC against number of days of lifetime, the Pearson correlation coefficients are $r = 0.9949$ for FreeBSD, $r = 0.9972$ for NetBSD and $r = 0.9944$ for PostgreSQL (significance test: $p < 2.2 \cdot 10^{-16}$ for the three cases). With these Pearson coefficients, regression models could be fitted, and I would probably obtain accurate results for predictions in the near future of the systems. However, as I

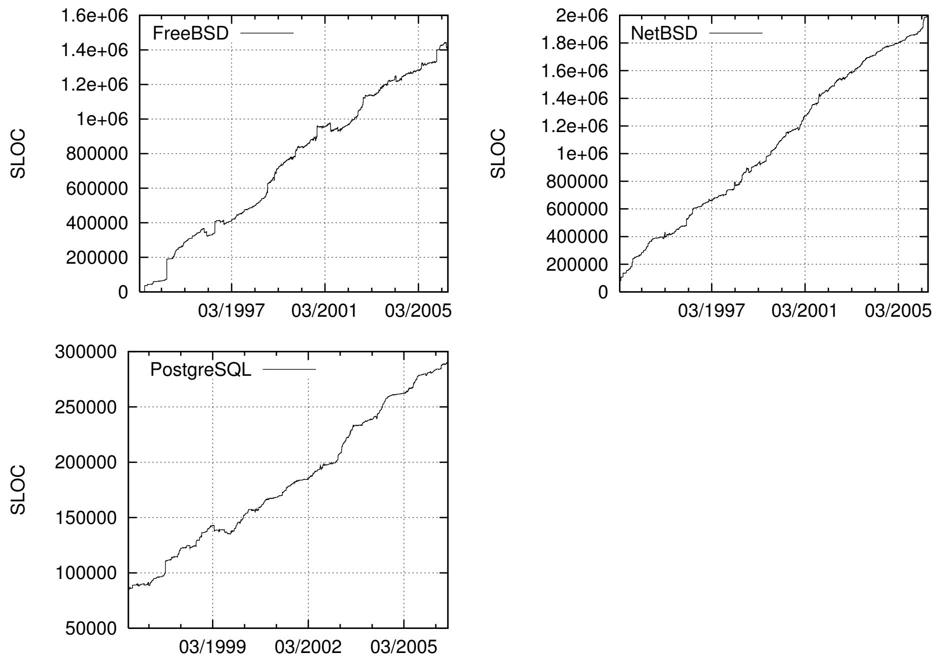

Figure 4.19: *Plots of SLOC over time (daily series) for NetBSD, FreeBSD and PostgreSQL.*

show at the end of this section, even with those good correlation coefficients, the time series model performs better than the regression model when predicting the last year of history of the three case studies.

4.7.1 Fitting ARIMA models

The first step to fit an ARIMA model for a time series, is to identify the values of the three parameters of the model: p, d and q[4]. As the linear regression coefficients suggest, the series were close to be linear. Therefore, I selected $d = 1$ to get stationary data. To select p and q, I needed to plot the autocorrelation and partial autocorrelation functions. When I tried to do so with the original data (this is, without any filtering), I did not obtain a clear pattern and I could not apply the criteria described in table 3.4 on page 62.

For instance, figure 4.20 shows the autocorrelation and partial autocorrelation coefficients for the first difference of the FreeBSD series; the coefficients are shown for the first ten lags, this is, it shows the internal autocorrelation among the last ten points of the series. The number of lags is not important, as long as enough lags are shown to find out the exact pattern of the plots. In the case of the autocorrelation coefficients function, only the first coefficient is significant, and after that lag, all coefficients suddenly drop to zero. In the case of the partial autocorrelation coefficients, all coefficients are very

[4]The p parameter here is different to the p statistical that we have used through this chapter to show the significance level of the Pearson correlation coefficients. We have chosen not to change the symbol, because the usage of p for both parameters is very common in the scientific literature

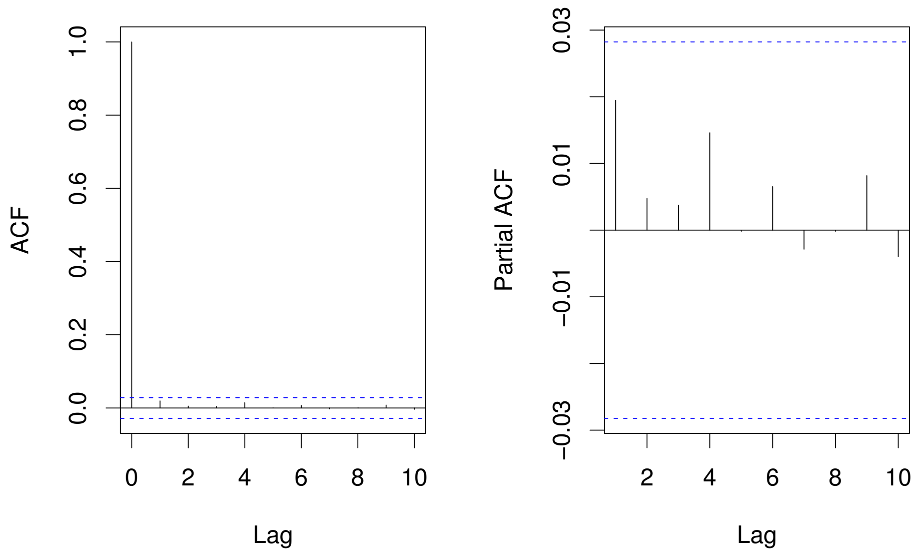

Figure 4.20: *Autocorrelation (left) and partial autocorrelation (right) coefficients for the first ten lags of the difference of the FreeBSD series (no smoothing applied).*

low, and all of them should be discarded attending to the criteria shown in table 3.4. Therefore, figure 4.20 shows that noise is reducing the internal autocorrelation of the series.

I needed therefore to filter the original series to remove the noise. Figure 4.21 shows the first difference of the FreeBSD series before and after applying a *kernel smoothing* process (the process was described in the previous chapter in section 3.3.2). The plot in the top of that figure is the original data, the plot in the middle is the smoothed series, applying a bandwidth of 10. The plot in the bottom is the smoothed series, with a bandwidth of 50. Smoothing removes some noise from the series, and the trend in the data appears now much clearer. I applied smoothing with a bandwidth of 50 to the series of the three case studies.

After the smoothing process, the pattern in the autocorrelation and partial autocorrelation functions was much clearer. Figure 4.22 shows those plots. According to table 3.4, I would choose $p = 9$ and $q = 0$, because the autocorrelation function slowly tails off to zero, and the partial autocorrelation function drops to zero after 9 lags.

In the case of NetBSD, the pattern was the same with 6 significant lags in the partial autocorrelation coefficients. Therefore $q = 0$ and $p = 6$. In the case of PostgreSQL, the pattern was the same with $q = 0$ and $p = 7$.

4.7.2 Accuracy of the models

To test the goodness of the obtained models, I split the series in two sets: the training set and the test set. The training set was built with all series but the last year. The last

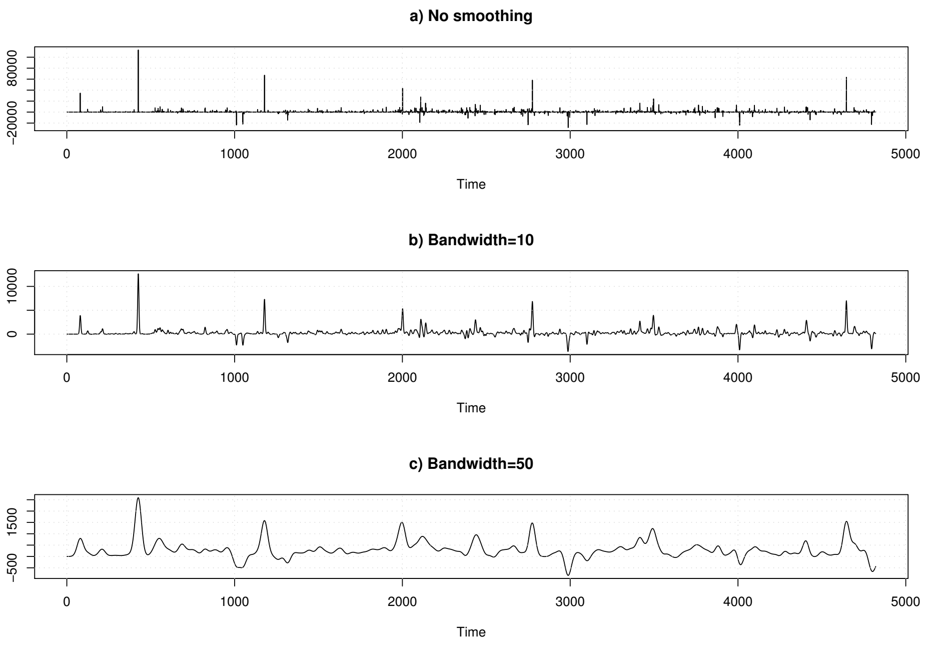

Figure 4.21: *Original series and smoothed series for FreeBSD*

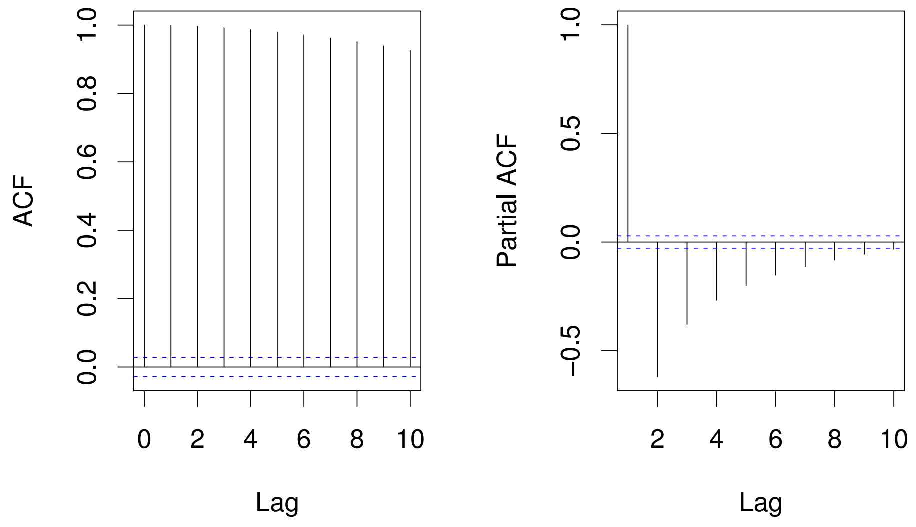

Figure 4.22: *Autocorrelation (left) and partial autocorrelation (right) coefficients for the first ten lags of the smoothed FreeBSD series (first difference).*

	ARIMA		Regression	
Case study	**MSRE**	**Sd. dev.**	**MSRE**	**Sd. dev.**
FreeBSD	3.93	3.28	16.89	14.82
NetBSD	1.80	1.28	15.94	8.65
PostgreSQL	1.48	1.86	6.86	4.75

Table 4.9: *Mean squared relative errors (MSRE) and standard deviation of the squared relative errors (Sd. dev.) for the time series and regression models.*

year was the test set[5]. The values obtained from the model were the first difference of the forecasted series. I made the inverse operation of the first difference, and I added the value of the actual series at the beginning of the test interval. Then I compared these forecasted values with the actual values.

After fitting the time series model using the training set, I predicted the next year, and I compared the forecasted values against the actual values contained in the test set. Table 4.9 contains the mean squared relative errors (MSRE) for the three case studies, both for the time series (ARIMA) and for the regression models.

I repeated the same procedure with a linear model obtained by statistical regression using least squares. I correlated SLOC against number of the days of lifetime, for all the data but the last year (this is, I used the training set for least squares regression). Then I compared the last year forecasted with the regression model against the actual values (the test set). For the case of PostgreSQL the Pearson correlation coefficient with the training set was $r = 0.9945$ for FreeBSD, $r = 0.9973$ for NetBSD and $r = 0.9928$ for PostgreSQL (significance test: $p < 2.2 \cdot 10^{-16}$ for the three cases). Table 4.9 also shows the mean squared relative errors for this model.

When comparing the mean squared relative errors for both models, we can easily see that the time series model is more accurate than the regression one. Therefore the time series based ARIMA models can predict the growth in the next year of a project with lower error than regression models. It is remarkable that even with the high correlation coefficients obtained with linear models, the ARIMA models performed better than the regression models.

4.7.3 Summary

In this section, I have shown the results of fitting time series models to predict the evolution of software systems. I used the Box-Jenkins method to fit ARIMA models, to the daily time series of size, for the cases of FreeBSD, NetBSD and PostgreSQL.

The growth curves of the three case studies resulted to be very close to linear. Because of that, I decided to compare the accuracy of the ARIMA models with linear regression models. To do so, I split the series in two sets: training and test. The training

[5]The test sets contained 321 days for FreeBSD, 326 days for NetBSD and 361 days for PostgreSQL. The number of days is almost one year for all the cases.

set contained all the data but the last year. After fitting the models, I calculated their accuracy comparing against the test set (this is, comparing with the actual last year of the case studies).

The ARIMA models had a memory of approximately one week for the three case studies. In spite of this short memory, the models could predict the next year of history with less than 4% of mean squared relative error in the case of FreeBSD, and less than 2% for the cases of NetBSD and PostgreSQL. For the same cases, the accuracy linear regression models, measured as MSRE, was $\sim$ 16% for FreeBSD and NetBSD, and $\sim$ 7% for PostgreSQL (with Pearson coefficients over 0.99 for the three regression models).

CHAPTER

FIVE

Conclusions and further work

5.1 Summary of results

In chapter 4, I have presented the results obtained in this thesis. The first result shows that size and complexity metrics are highly correlated, both at the file and at the product level. Therefore, all the metrics are providing the same information from a statistical point of view. The next result shows that the statistical distribution of the size of software is a double Pareto. That distribution appears at different levels of granularity (file, product and domain of application), which is an evidence of self-similarity in software. The next result shows that the dynamics of software evolution is a short range correlated process, what implies that ARIMA models are suitable for the statistical modelling of software evolution.

I analyze these results, and their theoretical and practical implications, in the next sections.

5.1.1 Correlation of metrics

In chapter 2 (section 2.3.4), I highlighted the conflict between some empirical studies of libre software with the classical works in software evolution. Basically, Lehman argued that the conflicting studies were not comparable because of the size metrics used, claiming that number of modules (this is, source code files) were a superior metric than lines of code.

The results of this thesis show that number of files is highly correlated with the rest of the metrics of size and complexity, and in particular, with SLOC (that is the metric used by the conflicting studies). In other words, from a statistical point of view, both SLOC and number of files are providing the same information. Considering that those studies obtained statistical models of software evolution, and that the conclusions were based on those models, I affirm that **the studies by Godfrey *et al.* [GT00] (and other conflict-**

ing works using SLOC [RAGBH05]) and Lehman [LB85] are comparable. Therefore, the Lehman laws are not valid for the cases studied in the mentioned works

5.1.2 Software attributes

The two main attributes that can be used to characterize a software product are size and complexity. The result discussed in the previous paragraphs shows that, for the case of the C programming language, all the size and complexity metrics considered in this thesis are highly correlated. This has been verified with different samples: an overall sample using all the files available in the sample, and subsamples that formed more homogeneous groups (dividing by file type, package size and field of application).

This result does not mean that the studied complexity metrics are not actually measuring complexity, but that those metrics have, from a statistical point of view, the same predictive power that other metrics like SLOC or LOC. Therefore, when constructing models which contain size or complexity terms (for instance, defect prediction models based on the attributes of source code), I recommend to use simple metrics like LOC or SLOC.

I have found two exceptions though. The first one regards to the Halstead's level metric. In some of the subsamples, that was the only metric not correlated with the rest. The second exception is header files. In that kind of files, cyclomatic complexity is not correlated with the rest of metrics. This is probably because of the nature of that kind of files. Header files are flat, neither containing bifurcations (like `if` statements) nor loops. Therefore, the cyclomatic complexity keeps low regardless the size of the header file.

As a practical conclusion, **for software evolution studies, I recommend to measure Halstead's level as well as SLOC**, and not to measure any other of the metrics studied here, because they have been shown to be highly correlated. In the case of header files, I recommend not to measure cyclomatic complexity, because of the nature of that kind of files. The correlations are verified regardless the size of the files, the size of the software systems to whom the file belongs, the domain of application of the system, and the nature of the file. Therefore, we suggest that the correlations that I have found is an universal property of software written in C.

5.1.3 Statistical properties of software

Much research has been devoted to the issue of the statistical distributions found in software systems, and more specifically, to the presence of power laws in the distribution of some of the properties of software systems. From all the works that were reviewed in chapter 2, I highlighted the work by Wu in his PhD Thesis [Wu06] (see section 2.4.3). He found power laws in the distribution of the size of the system (measured as number of developers, and number of commits), and long range correlations in the time series of number of changes.

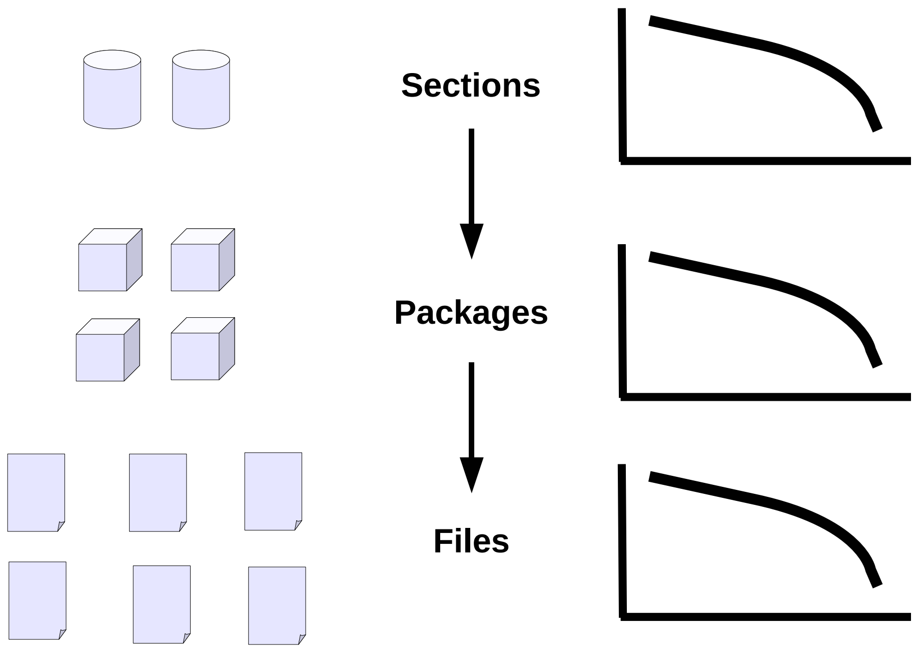

Figure 5.1: Self similarity in software. The size of source code files, the size of software packages, and the size of sections have all the same statistical distribution.

In this thesis, I have estimated the statistical distributions of size and complexity using a large sample of software products. I have found that the distributions are *double Pareto*. This is in fact a family of distributions, originally proposed by Reed and Jorgensen [RJ04], that has been proved useful for modelling size distributions in a wide range of phenomenons (incomes and earnings, human settlement sizes, oil-field volumes, particle sizes). Mitzenmacher [Mit04b] found this distribution in file systems, using the size of files. He proposed a model (the *Random Forest File Model*) to explain and simulate the process that generates this distribution in file systems. In section 5.2, **I review the *Random Forest File Model*, and provide guidelines to adapt it to the case of software evolution** (although I leave that task for further work).

The same distribution appears regardless the level of aggregation used. Figure 5.1 shows this. If we measure the size of source code files, the distribution of size is double Pareto. If we measure the size of software packages (using number of files that each package contains), the distribution of that parameter is also double Pareto. If we measure the size of sections (using the number of packages that each section contains), the distribution of that parameter is again double Pareto

Therefore, regardless the level of aggregation (or scale) used, the distribution of size has the same shape. In other words, I have found that **there is self-similarity in software**. Self-similarity is commonly found in natural structures. Self-similar structures can evolve in size through some orders of magnitude. As it was highlighted in the fist chapters, libre software has been growing at high rates in the last years (Linux is growing with a super-linear profile, the Debian GNU/Linux double its size in half of the time compared to previous releases). With such an evolutionary pattern in size, it is natural to obtain self-similar geometries. Other geometries would not support such a fast evolution. Clearly, this is a topic that deserves further research.

5.1.4 Short memory dynamics and ARIMA models

In his PhD thesis, Wu found power laws distribution in the size of changes, and long range correlations in the time series of changes [Wu06]. The first finding (power laws in the size of changes) is an evidence of self-similarity. The second finding (long range correlations in time series), together with self-similarity in size, are evidences of self-organized criticality in the system (a review of these concepts is given in section 2.4.3).

As shown above, I have verified the first finding, although using a different approach. However, I could not verify the second finding. As shown in the previous chapter, in a sample of 3, 821 projects, I have found a **short range correlated evolution dynamics**. This has been verified using the time series of the number of commits and the number of modification requests (with a period in the series of one day). Therefore, this finding would contradict that software evolution is driven by a self-organized criticality dynamics.

To manage, forecast and control software evolution, I propose to use ARIMA models. That kind of models are short memory: only values in the near past of the time series are influencing the current value of the series. In the previous chapter, I have shown that **ARIMA models can predict with accuracy the evolution of software systems**. In the cases shown in the Results chapter, I had three systems, with linear profiles ($r > 0.99$). With those highly linear profiles, ARIMA models performed much better than linear regression models. The models had approximately a memory of one week. Another proof of the accuracy of this kind of models is that the winner of the MSR Challenge 2007 was an ARIMA model [HGBR07a]. The challenge consisted in predicting the number of changes in the CVS repository of Eclipse during the three months right after the deadline for submissions.

Summarizing, software evolution is driven by a short memory dynamics, and because of that, ARIMA models are a good option to forecast, manage and control software evolution.

5.1.5 Practical implications

The results obtained in this thesis have practical implications about software measurement and about empirical studies of software evolution. In more detail, those implications are the following:

- For the case of libre software written in C language, the conclusions of an empirical study on software evolution should be the same regardless the size metric used. This supposes that studies using different size metrics are comparable.

- Again when measuring C source code, in order to characterize a software product, bounded to the case of libre software, only Halstead's level is not highly correlated with size metrics. I recommend therefore to measure only SLOC and Halstead's level.
- The size of software is distributed like a double Pareto distribution. Because of this, the Mitzenmacher's *Random Forest File Model* (RFFM) can be used to simulate the evolution of a software product.
- The shape of the statistical distribution of the size of software is the same, regardless of the scale used (file, module, project). That is an evidence of self-similarity in software, which implies that the RFFM can be used to simulate evolution at different levels of granularity.
- Software evolution is driven by a short memory dynamics. Because of this, ARIMA models can be used to obtain accurate statistical models of the evolution of software systems.

I remark that some of the conclusions are bounded to the case of libre software written in C language because the sample used for the studies only contained products fulfilling those characteristics. However, at a first glance, there is no reason to think that the first three points shown above (comparability of studies, correlation of metrics and statistical distribution) are specific to the case of libre software.

5.2 Further work

The results and implications of this thesis have raised new research questions, that deserve further research. The statistical distribution of the size of software can be used to determine which kind of processes have generated the software product. In other words, it can be used to obtain models that simulate the behavior of the software product. This kind of models can be the base for a theory of software evolution. Of great interest is also that I have found self-similarity in software. This fact can be an explanation of the fast growth of some libre software projects, and it is clear that a theory for software evolution should take this finding in account. Finally, another fact that influences the design of this theory is that software evolution is driven by a short memory dynamics.

Although in this section I discuss all these topics in more detail, I leave them as open questions for further research.

5.2.1 The Random Forest File Model applied to the case of software evolution

I can not determine why the distribution of the size of software is a double Pareto, but that finding can be used to design models to simulate the evolution of software. For the more general case of file-systems, Mitzenmacher also found double Pareto distributions, and he proposed a model that could be used to simulate the dynamics of a file-system: the *Random Forest File Model* (RFFM) [Mit04b]. The purpose of this model was helping to manage file systems, in order to find out when more capacity is needed. However, that model is very close to the concepts found in software evolution. Because of that, the model could be adapted to simulate the dynamical behavior of a software product.

The nature of the model is probabilistic. For instance, two runs of the model over the same case study will produce two different results. Hence, instead of obtaining an only future scenario for the project, many future scenarios can be obtained. If the model is run several times, we could obtain a portfolio of scenarios, with a probability assigned to each of them.

Besides producing double Pareto size distributions, the model has another interesting property: the average depth of the source code trees generated by the model keeps bounded by a constant. This has been found in libre software systems before [CMR04b].

The RFFM shares some analogies with the Maintenance Guidance Model (MGM), that was reviewed in section 2.4.4. The MGM [CFR07] considers that developers choose the parts where they want to work on based on the previous activity of those parts. In particular, the parameter that the MGM considers is number of previous changes. If the number of changes is equal, the MGM considers that developers prioritize those parts with higher complexity, because the rewards of the work will be higher.

The MGM is like a deterministic version of the RFFM. There are two interesting points about this relationship between the MGM and the RFFM:

- The MGM has been empirically validated, and has performed well when compared against the real history of libre software systems.
- Although based on the statistical properties of the distribution of changes, the MGM has been obtained by reasoning how developers choose where to work on in a software system. It is interesting that this model has resulted to be so similar to a purely statistics-based model like the RFFM.

Summarizing, the following facts suggest that the RFFM could be a good candidate to simulate the evolution of a software system, at the file level:

- Software size is distributed like a double Pareto. The size of the source code trees generated by the RFFM, are distributed like a double Pareto.
- The depth of the generated trees is bounded by a constant. This property has been found in empirical studies of some libre software systems [CMR04b].
- The RFFM shares many commonalities with the MGM, that has been empirically validated against real cases [CFR07]. The RFFM is obtained exclusively using statistical considerations. However, the MGM is based on assumptions on how developers work, that were deduced partially using the Lehman's laws of software evolution.

5.2.2 Self-similarity influence on the evolutionary patterns of software

In section 1.2, I talked about the fast growth of libre software, and the influence of that fact in research. To recall what I showed there, the Linux kernel is growing at a *superlinear* rate, this is, its growth rate increases with time. The *superlinearity* increases with time as well. I also mentioned the case of Debian GNU/Linux. In 2002, Debian released the version 3.0 of the system, that contained 100 millions of SLOC. The initial release of Debian was in 1993. Therefore, the libre software community wrote at least 100 millions of SLOC, in about 10 years. The next release of Debian, version 3.1, was launched in 2005, three years after the previous release. The size of this new release was 200 millions of SLOC. Therefore, in only three years, the libre software community had written at least as many code as in all the previous years.

Debian and Linux continue to be under active development, and the projects do not seem to show any symptom of exhaustion. How can these software projects maintain their coherence while growing so fast? In natural phenomena, self-similar structures allow growth in size and complexity without loosing coherence. In software systems, I have observed that there is self-similarity in some of their properties.

Therefore, I have both findings: fast growth while maintaining coherence, and self-similarity. Whether the self-similarity of software is a consequence of its fast growth, or the other way around, remains as an open question.

5.2.3 Determinism and evolution

If software evolution is driven by a short memory dynamics, that would imply that software evolution is a non-deterministic process. In this section, I present what this means for the software development and maintenance processes.

Figure 5.2 represents a tree of all the possible events that may happen in a software project. The vertical dimension represents time. The top node is the initial situation of the project. At each step, a decision must be taken. That decision open some paths, and discard some others. After some steps, the consecutive decisions has driven the project to a particular situation, while it has discarded some other situations.

For instance, in the case of the figure, after 5 steps the project can end in a set of 4 different situations. Let us suppose that in the first decision the project follows the path on the right. With the events tree shown in the figure that would discard the situations $P1$, $P2$ and $P3$. In other words, once a decision is taken, that influences the future of the project, making it impossible to reach the states $P1$, $P2$ or $P3$. That is because there are not too many connections or paths between the nodes. The tree shown in figure 5.2 is a typical tree for a long memory process. Decisions taken long time ago have a heavy influence on what is going on today in the project.

Consider now the tree shown in figure 5.3. This time the tree has a much higher density of connections. Decision taken in the first step do not discard any possible future scenarios after some steps. For instance, after 5 steps there are 6 possible scenarios. Early

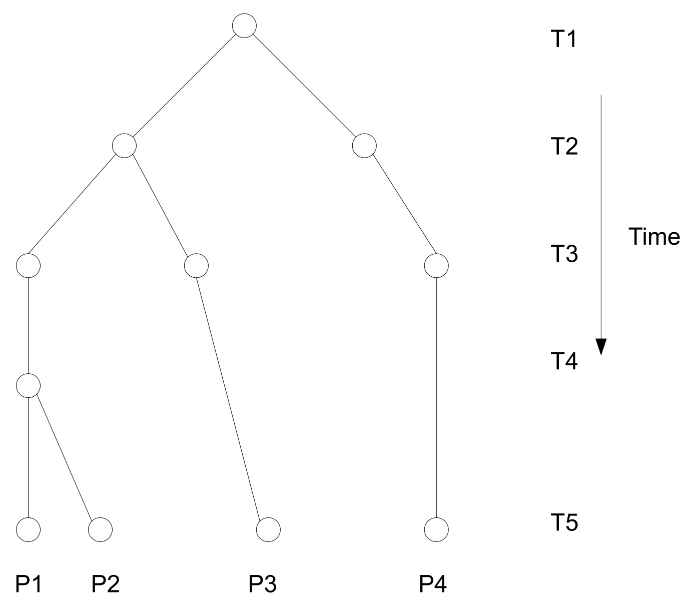

Figure 5.2: *Events tree. This tree represents all the possible events that may happen in a software evolution project. The top node is the initial situation. In each step, a decision is taken, and that implies to follow a particular path.*

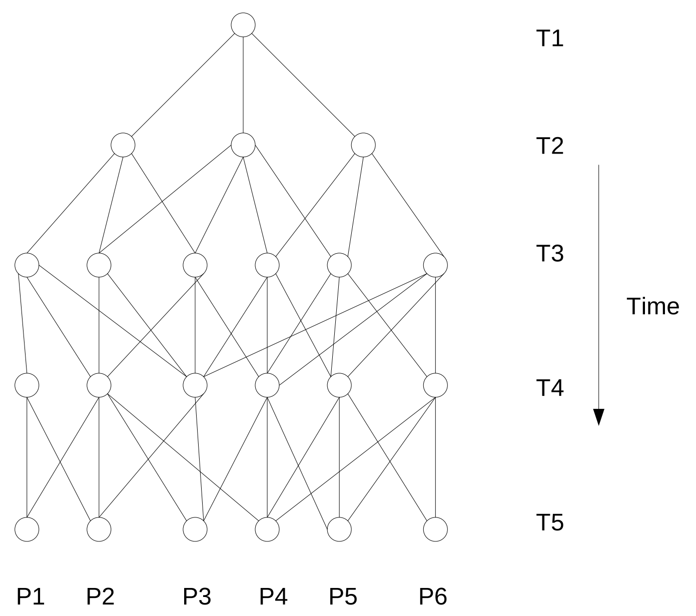

Figure 5.3: *Events tree for a short memory process. The density of connections is much higher, and so early decisions do not discard possible future scenarios.*

decisions do not discard any of those situations, and only recent events influence which scenario will finally take place. This tree corresponds to a short memory process.

However, I have not investigated the development process in order to find out if the dynamics is like the long memory tree shown in figure 5.2 or like the short memory tree shown in figure 5.3. I intend to do so as further research.

In any case, this finding does not imply that there are not long term effects in software evolution. For instance, there are periodical events. In some sense, a model for software evolution should be like a model for weather forecasting. Weather today can be predicted based on weather in the last days, but also in the season of the year. For instance, if it is summer and today is sunny, it is likely that tomorrow will be sunny (short term effect), and it is very unlikely that will snow because it is summer (periodical effect). However, we can not say if next week will be sunny, because the memory of the process is very short as to make predictions with that time window.

I leave the investigation of periodical events, and the development of a model like the one described above, as further research.

5.3 Conclusions

This thesis is an empirical study of the evolution and properties of software, bounded to the case of libre software. It is completely based on publicly available data sources. The analyses performed in this thesis are based on statistical techniques, and in particular, on time series analysis for the study of software evolution.

It addresses some questions that have been subject of controversy in the research community. The classical theories for software evolution [LB85] were faced with the born of libre software development, and soon many conflicting cases started to appear [GT00, RAGBH05]. However, the methodologies used in the conflicting cases were not exactly the same that in the classical studies on software evolution. The comparability of those studies with the classical works was discussed in the literature [LRS01].

These works fostered the research on the topic of software evolution and libre software, and some proposals for the dynamics of libre software have appeared [Wu06, WHH07]. In particular, it has been proposed that the mechanisms of the evolution of libre software are a Self-Organized Criticality (SOC), because there is self-similarity in software, and its evolution is a long range correlated process.

Thanks to the availability of large public datasets, I have been able of verifying the mentioned discrepancies in a large number of case studies. I have found that, for the case of libre software written in language C, software evolution studies can be compared even if they use different metrics. For the same cases, most of the size and complexity metrics are highly correlated, which implies that statistical models using those attributes can be simplified. I have found that the dynamics of software evolution is not SOC, because although I have found self-similarity, I have determined that the evolution of libre software is a short range correlated process.

While addressing those questions, I have also determined that the distribution of the size of software is a double Pareto. There is self-similarity in software because this distribution appears at different scales (file, package and field of application levels).

To address the first questions (comparability of studies, correlation of metrics), I measured the size and complexity of a sample of 6,556 products obtained using the FreeBSD *ports* (or packages) system. I performed a correlation analysis over that sample of files, and over some subsets of more homogeneous groups (discriminating by file type, package size and field of application). Using another sample of 12,010 products also obtained from FreeBSD, I calculated some statistical properties, and found the double Pareto distribution.

For the rest of questions (kind of dynamics), I have used a sample of 3,821 projects obtained using the FLOSSMole and CVSAnaly-SF datasets. I obtained the time series of daily number of changes (both in number of commits and in number of modification requests). I applied time series analysis to determine that those series were short range correlated. Because of that, I have shown that ARIMA models (a kind of models typical in time series analysis) perform better than regression models to forecast software evolution.

All the results of this thesis are easily verifiable, because all the databases and tools used to obtain them are publicly available. That should also help to verify if the conclusions of this thesis hold for other case studies.

The approach used in this thesis has been to perform a statistical analysis on a large dataset, to determine whether I could find patterns and commonalities among the different case studies under consideration. It might seem a novel approach, because it is possible thanks to the availability of thousands of software projects (and datasets for research like FLOSSMole) in the libre software community. However, as early as 1974, Lehman suggested to use this approach [Leh74]. He even suggested to apply time series analysis to the case of software evolution. Unfortunately, that approach could not be done at that time because of the lack of databases and indexes of software projects.

In any case, to my knowledge, and despite the suggestions by Lehman, this statistical approach has not been addressed at such a large scale to date. The problem with small scale empirical studies is that it is difficult to determine whether or not the conclusions hold for other cases, which causes the kind of controversies like the one about the evolution of libre software. Large scale studies ensure statistical significance, and what is more important, statistically significant findings can be considered universal properties. Thus, I propose that all the findings reviewed and discussed in this thesis are universal properties of software (or at least, of libre software for some findings, and for the same kind of software written in C language for others).

A theory for software evolution should be designed considering these universal (or statistically significant) findings as starting points. That should help to avoid future controversies about the validity of some laws to some particular case, and it also should help to develop the necessary scientific basis for Software Engineering.

APPENDIX

A

The double Pareto distribution

This appendix briefly describes the properties and equations of the double Pareto distribution. This distribution is a combination of one or two power law tails, and a lognormal body.

The power law distribution has a cumulative distribution function as follows:

$$\Pr[X > x] = \left(\frac{x}{k}\right)^{-\alpha} \tag{A.1}$$

with $\alpha > 0$, $k > 0$ and $X \geq k$, and where k is a constant, and α is the scaling parameter (or exponent) of the power law.

The density function of a power law distribution is:

$$f(x) = \alpha k^{\alpha} x^{-\alpha-1} \tag{A.2}$$

Both the density function and the cumulative distribution function appear as straight lines in logarithmic scale. For more information about the power law distribution, its shape, and how to fit a power law to empirical data, I recommend the paper by Clauset *et al.* [CSN07].

The lognormal distribution appears when the logarithm of a variable is normally distributed. This is, if X is lognormal, then $Y = log(X)$ is normal. Therefore, the density function and the cumulative distribution function will be the same that in the case of normal distribution (with a variable substitution).

The density function is therefore

$$f(y) = \frac{1}{\sqrt{2\pi}\sigma y} \exp -\frac{(\ln y - \mu)^2}{2\sigma^2} \tag{A.3}$$

with $y = log(x)$, and where μ is the mean of the distribution and σ the standard deviation.

The cumulative distribution function is

$$\Pr[Y > y] = \int_{z=y}^{\infty} \frac{1}{\sqrt{2\pi}\sigma z} \exp - \frac{(\ln z - \mu)^2}{2\sigma^2} dz \tag{A.4}$$

The double Pareto distribution can be constructed using the above equations. The density function for the high values end is as follows:

$$f_1(x) = \alpha x^{-\alpha-1} A(\alpha, \nu, \tau) \phi \left(\frac{\log x - \nu - \alpha\tau^2}{\tau} \right) \tag{A.5}$$

While the low values tail has a density function as follows:

$$f_2(x) = \beta x^{\beta-1} A(-\beta, \nu, \tau) \phi^c \left(\frac{\log x - \nu + \beta\tau^2}{\tau} \right) \tag{A.6}$$

In the above equations ϕ is the cumulative distribution function of the standard normal distribution, ϕ^c is the complementary of ϕ, and

$$A(\theta, \nu, \tau) = \exp \left(\frac{\theta\nu + \alpha^2\tau^2}{2} \right) \tag{A.7}$$

The density function of the double Pareto can be written then as follows:

$$f(x) = \frac{\beta}{\alpha+\beta} f_1(x) + \frac{\alpha}{\alpha+\beta} f_2(x) \tag{A.8}$$

The cumulative distribution function is as follows:

$$\begin{aligned} \Pr[X > x] &= \phi \left(\frac{\log x - \nu}{\tau} \right) - \\ &- \frac{\beta}{\alpha+\beta} x^{-\alpha} A(\alpha, \nu, \tau) \phi \left(\frac{\log x - \nu - \alpha\tau^2}{2} \right) + \\ &+ \frac{\alpha}{\alpha+\beta} x^{\beta} A(-\beta, \nu, \tau) \phi^c \left(\frac{\log x - \nu + \beta\tau^2}{2} \right) \end{aligned} \tag{A.9}$$

In this thesis, I use the complementary cumulative distribution function (CCDF) in logarithmic scale to determine whether an empirical distribution is a double Pareto or not. The CCDF of a double Pareto has two straight tails, corresponding to the power law parts, and a lognormal body that connect the two tails. In some cases, the low values tail can be modelled using a lognormal, because the difference between a lognormal and a

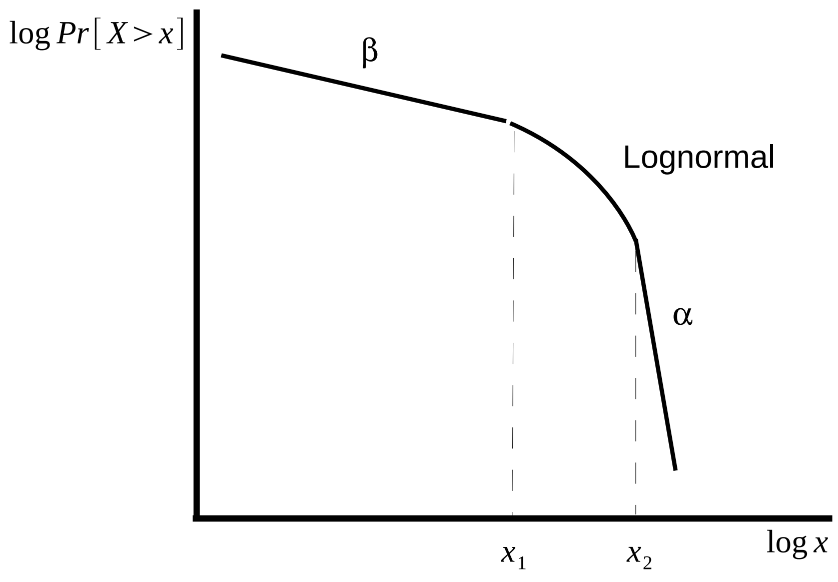

Figure A.1: *Complementary cumulative distribution function of a double Pareto. The plot shows two straight tails, corresponding to power law distributions with scaling parameter α and β, and a lognormal body that connects the two tails.*

power law in that region is very small. Appendix B includes some plots that show how that tail is very close to a lognormal in some of the cases (for instance, the case of the HLEVE metric in figure B.1).

Figure A.1 shows the ideal case of a CCDF of a double Pareto distribution. The low value tail (power law with scaling parameter β) changes to a lognormal body in the point x_1. The lognormal body changes to a straight tail (power law with scaling parameter α) in the point x_2. In this thesis, I have fitted the empirical data to a lognormal for all the values below x_2, and to a power law for the values greater than x_2. Appendix B includes some tables with the parameters of the lognormal body and the power law tail (in those tables, x_{min} corresponds to the value x_2 shown in figure A.1).

The double Pareto distribution was first described by Reed and Jorgensen [RJ04]. Mitzenmacher based on that distribution to obtain a model of the dynamics of file-systems, and includes a description of the properties of this distribution [Mit04b].

APPENDIX

B

Additional results

In chapter 4, I have shown some statistical properties of the sample of source code, and that the size of those source code files can be described using a double Pareto distribution. In this chapter, I show some additional results about those properties.

Figure 4.10 has shown that the empirical data can be described using a lognormal distribution for the lower values, and a power law for the higher values. In this chapter, I have fitted the same type of distributions to all the metrics, for three different samples: all the files, only header files, and only non-header files.

Figure B.1 shows the results for the sample of all the source code files. The plot shows the empirical data, the lognormal distribution with the same mean and standard deviation that the sample, and a power law fitted using the procedure suggested by Clauset *et al.* [CSN07]. Some of the plots (HVOLU, HLEVE) seems to have a clear power law tail for the low values, as well as the power law for the high values tail. However, the lognormal distribution also fitted well in that region. Therefore, it is difficult to determine if that region is indeed a power law or a lognormal distribution. In any case, all the cases fit well with the combination of a lognormal for low values and a power law for high values. Table B.1 shows the values of the parameters of the double Pareto distribution, for all the metrics. It includes the mean ($\bar{x}$) and standard deviation (s) for the lognormal part (this is, the mean and standard deviation of the logarithm of the metric), the transition point where the lognormal and power law ends connect (x_{min}), and the power law exponent (α).

Figure B.2 shows the same plots for the sample of only header files. For some of the metrics (for instance, FUNCS), the power law character of the data seems to have more influence than the lognormal part. Again, the Halstead's metrics seem to present the lower power tail more clearly than the rest of metrics. In any case, all the metrics seem to fit well to a combination of a lognormal and a power law distributions. Table B.2 shows the parameters of the double Pareto distribution for all the metrics.

Finally, figure B.3 shows the cumulative distribution function plot for all the metrics, for the case of non-header files. This sample is very similar to the overall sample shown in figure B.1. Again, the Halstead's software science metrics seem to present two

Metric	$\bar{x}$	s	x_{min}	α
SLOC	4.445	1.637	3251	3.34
LOC	5.044	1.404	5394	3.50
FUNCS	1.257	1.240	83	3.36
CYCLO	2.088	1.930	580	3.33
HLENG	6.075	1.759	5465	2.58
HVOLU	7.903	2.016	57369	2.72
HLEVE	−3.496	1.271	.010133	2.52
HMD	11.399	3.204	$43.091 \cdot 10^6$	2.27

Table B.1: *Parameters of the double Pareto distribution, for all the metrics using the overall sample. The first column shows the mean of the lognormal part ($\bar{x}$). The second column shows the standard deviation of the lognormal part (s). The third column shows the value of the transition point from lognormal to power law (x_{min}). The fourth column shows the value of the power law exponent (α).*

Metric	$\bar{x}$	s	x_{min}	α
SLOC	3.415	1.355	803	2.57
LOC	4.281	1.127	797	2.71
FUNCS	0.0888	0.429	7	2.17
CYCLO	0.243	0.762	49	2.44
HLENG	5.008	1.505	19651	2.91
HVOLU	6.707	1.782	6836	2.04
HLEVE	−2.459	0.953	.100	2.46
HMD	9.166	2.607	$1.237 \cdot 10^6$	1.77

Table B.2: *Parameters of the double Pareto distribution, for all the metrics using the sample of header files. The first column shows the mean of the lognormal part ($\bar{x}$). The second column shows the standard deviation of the lognormal part (s). The third column shows the value of the transition point from lognormal to power law (x_{min}). The fourth column shows the value of the power law exponent (α).*

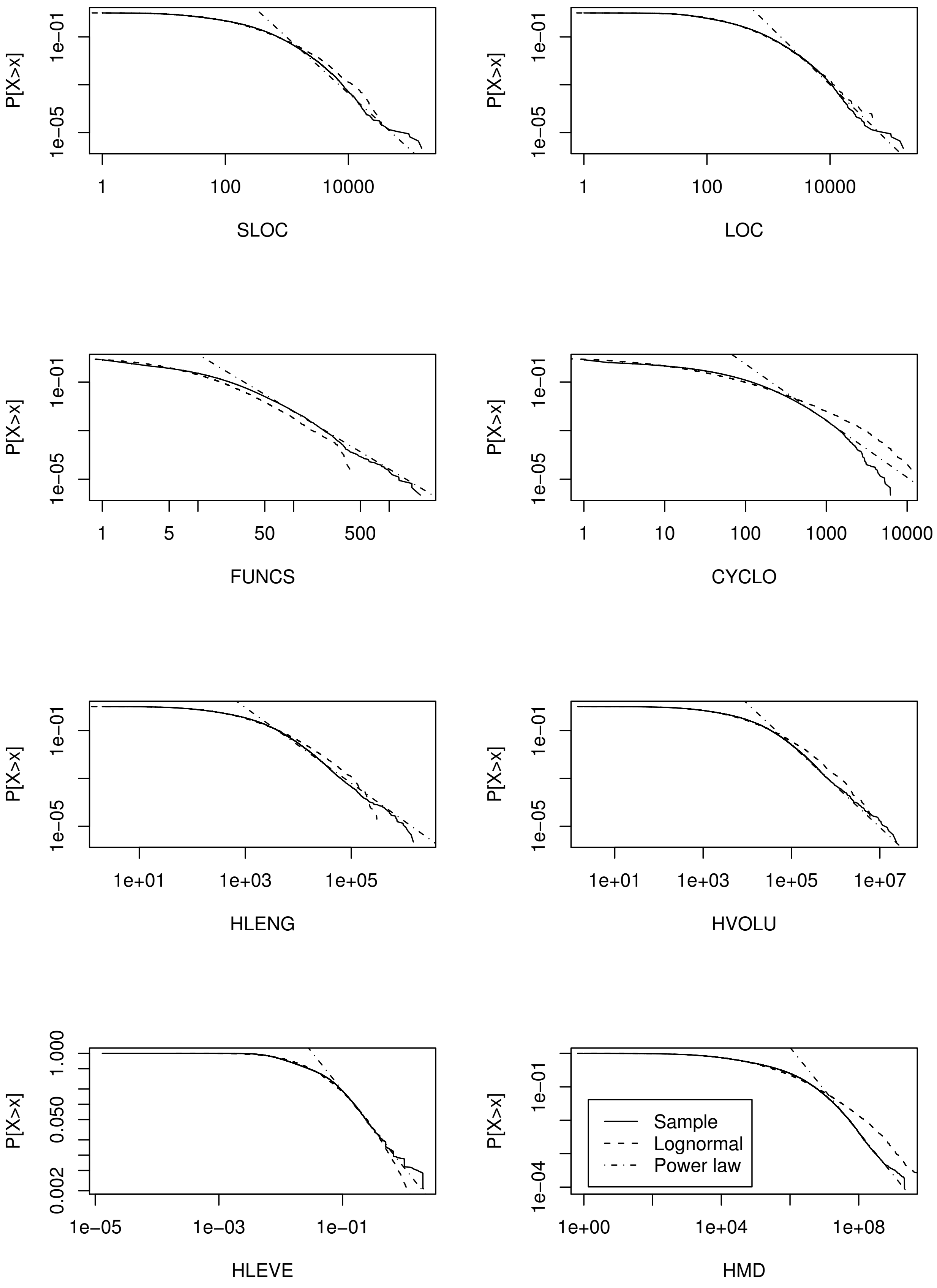

Figure B.1: *Complementary cumulative distribution function for the sample of all (header and non header) files. This plot shows the distribution plots for all the metrics. The legend is the same for all the plots. The plots show a lognormal and a power law fitted to the two extremes of the data.*

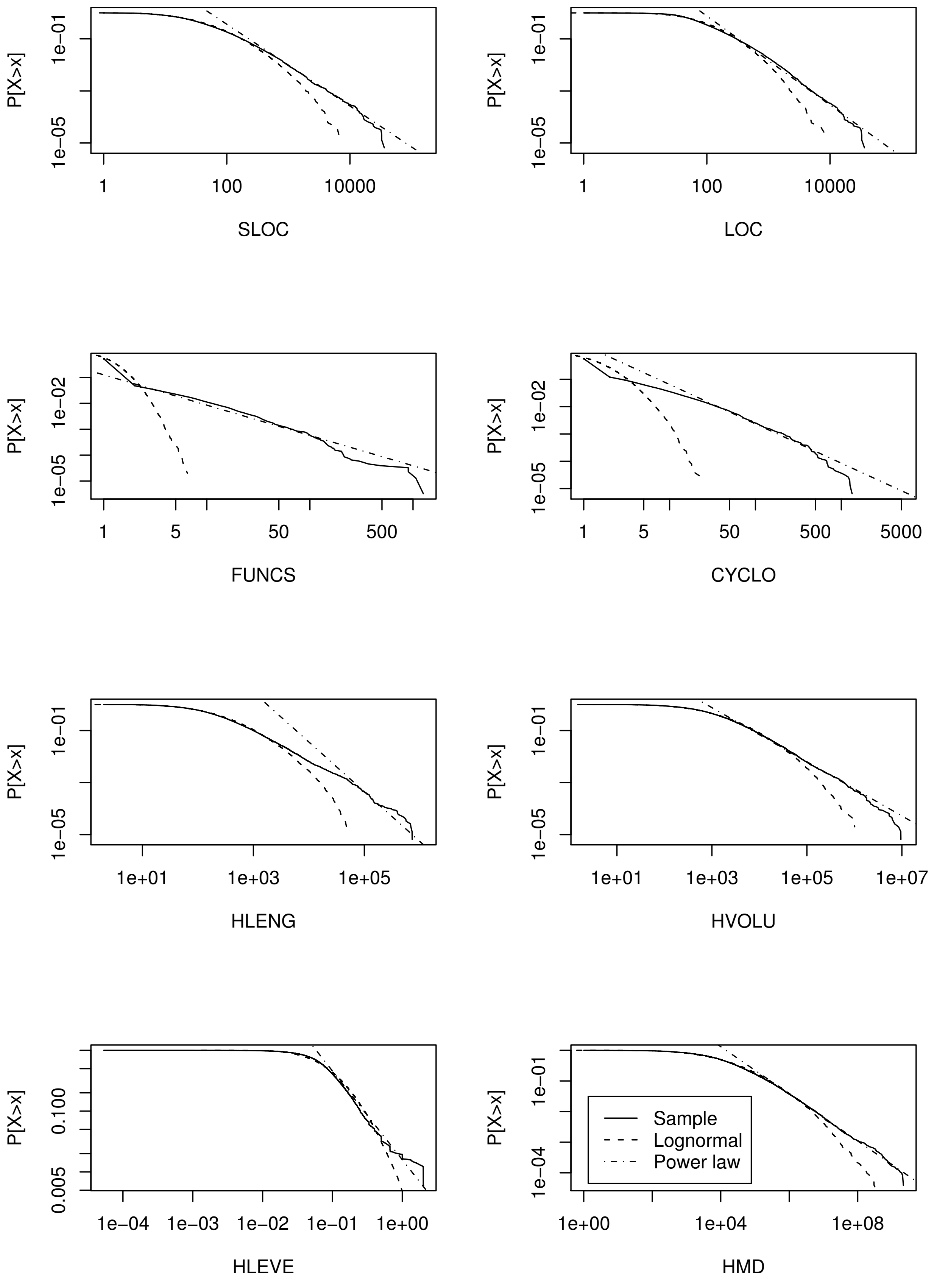

Figure B.2: *Complementary cumulative distribution function for the sample of header files. This plot shows the distribution plots for all the metrics. The legend is the same for all the plots. The plots show a lognormal and a power law fitted to the two extremes of the data.*

Metric	$\bar{x}$	s	x_{min}	α
SLOC	4.979	1.510	1221	3.00
LOC	5.451	1.354	1697	3.40
FUNCS	2.072	0.932	40	3.02
CYCLO	3.046	1.644	580	3.33
HLENG	6.629	1.621	7818	2.72
HVOLU	8.524	1.844	18723	2.92
HLEVE	-4.034	1.065	.083	2.46
HMD	12.557	2.853	$58.580 \cdot 10^6$	2.41

Table B.3: *Parameters of the double Pareto distribution, for all the metrics using the sample of non-header files. The first column shows the mean of the lognormal part ($\bar{x}$). The second column shows the standard deviation of the lognormal part (s). The third column shows the value of the transition point from lognormal to power law (x_{min}). The fourth column shows the value of the power law exponent (α).*

power law tails, although the low values tail can be also described using a lognormal distribution. Table B.3 shows the parameters of the double Pareto distribution for all the plots.

Summarizing, the distribution of the size and complexity of source code files is a double Pareto. Some metrics seem to present two power law tails connected by a lognormal body, while others present only power law tails for high values. In any case, the low values power law tails can be also described using a lognormal distribution. The distribution appears both for header and non header files.

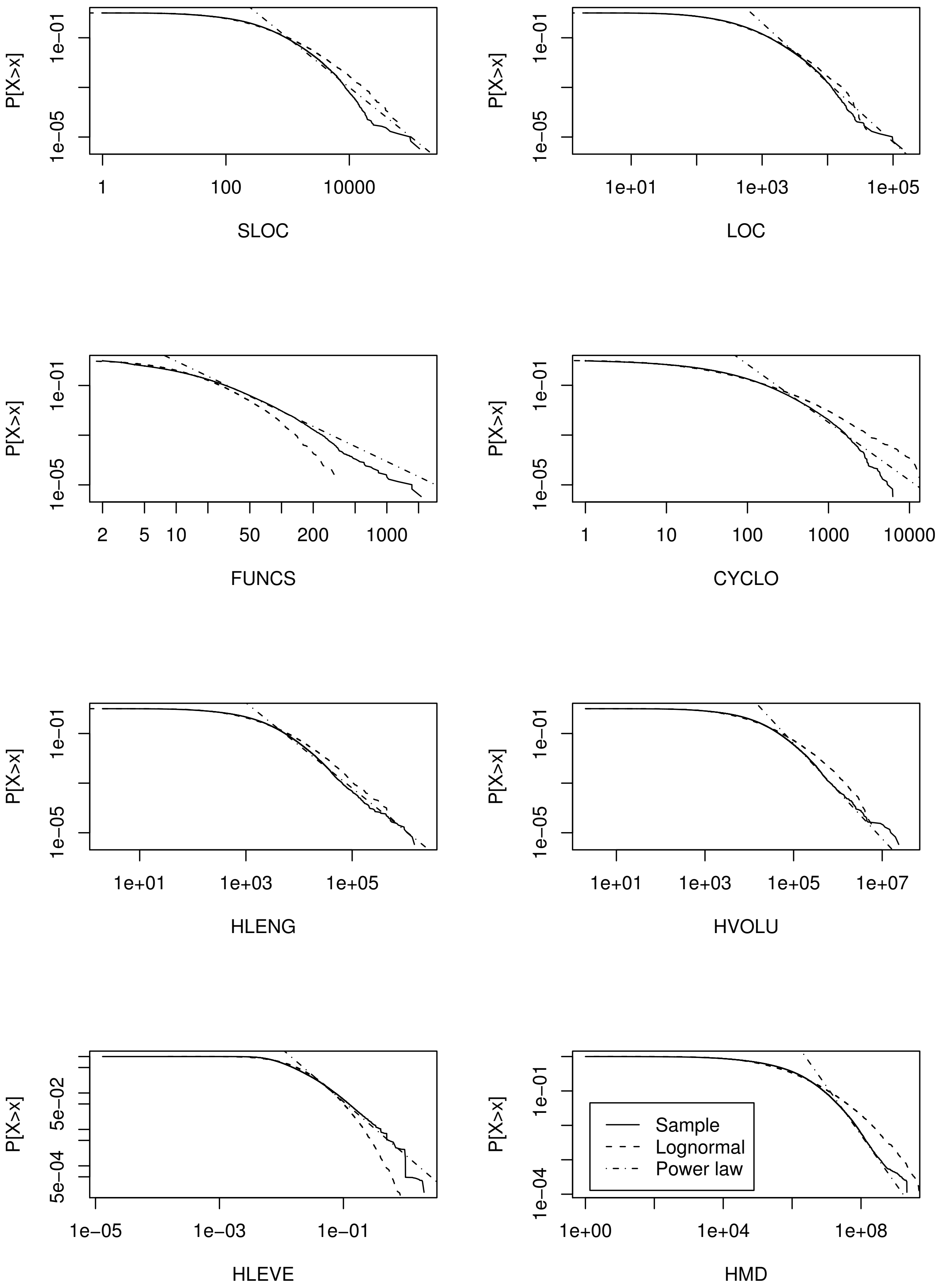

Figure B.3: *Complementary cumulative distribution function for the sample of non-header files. This plot shows the distribution plots for all the metrics. The legend is the same for all the plots. The plots show a lognormal and a power law fitted to the two extremes of the data.*

APPENDIX

C

Time Series Analysis functions and models

This appendix develops some of the time series analysis concepts that I have used during this thesis. For full details about the functions and concepts shown here, I refer to [SS06] and [MWH98].

C.1 Autocorrelation Coefficients Function

Let x_t be a time series of values $\{x_{t1}, x_{t2}, \dots, x_{tn}\}$. The mean function $\bar{x}$ is defined as follows:

$$\bar{x} = \frac{1}{n} \sum_{t=1}^{n} x_t \tag{C.1}$$

The autocovariance function γ is defined as follows:

$$\gamma(k) = n^{-1} \sum_{t=1}^{n-k} (x_{t+k} - \bar{x})(x_t - \bar{k}) \tag{C.2}$$

with $\gamma(k) = \gamma(-k)$ and $k \in [0, n-1]$

The autocorrelation function (ACF) r is defined as follows:

$$r(k) = \frac{\gamma(k)}{\gamma(0)} \tag{C.3}$$

This function fulfills the property $-1 \leq r(k) \leq 1$.

If x_t is white noise, $r(h)$ is normally distributed, with $\mu = 0$ and $\sigma = \frac{1}{\sqrt{n}}$. This property can be used to discard coefficients that are not significant. For instance, if the value of $r(h)$ falls within the following interval, the hypothesis of the coefficient being null is true with a 95% significance ($p = 0.05$):

$$\frac{-2}{\sqrt{n}} \leq r(k) \leq \frac{2}{\sqrt{n}} \tag{C.4}$$

C.2 ARIMA models

This section shows the equations of an ARIMA model. Those are the equations that have been used in this thesis to forecast the evolution of software projects.

Let us define the backshift operator B as follows

$$B^i_{x_t} = x_{t-i} \tag{C.5}$$

We can make algebraic operations with B. For instance:

$$x_t - x_{t-1} = \bigtriangledown x_t = (1-B)x_t \tag{C.6}$$

An ARIMA model has two linear components, as shown in the following equation:

$$\bigtriangledown^d x_t \left(1 - \sum_{i=1}^{p} \phi_i B^i\right) = \epsilon(t) \left(1 - \sum_{j=1}^{q} \theta_j B^j\right) \tag{C.7}$$

where d, p and q are the parameters of the model (see Results chapter for details about how these parameters are obtained), ϕ and θ are the coefficients that are obtained by fitting procedures, and ϵ are the estimation errors (with $\epsilon(1) = 0$).

C.3 Long and short memory processes

Another formula for the autocorrelation coefficient is the following:

$$r(k) = \frac{\Gamma(k+d)\Gamma(1-d)}{\Gamma(k-d+1)\Gamma(d)} \tag{C.8}$$

where Γ is the gamma mathematical function.

In a long memory process, that equation is proportional to k^{2d-1}. In other words

$$\log r(k) \sim C + (2d-1)\log k \tag{C.9}$$

where C is a constant and k is the time lag.

If $0 < d < 0.5$, then

$$\sum_{-\infty}^{+\infty} \mid r(k) \mid = \infty \tag{C.10}$$

and hence the *long memory* qualification.

For short memory processes[1], the ACF is proportional to k.

Therefore, those two properties can be used to distinguish among long and short memory processes.

[1]Actually, for integrated ARIMA processes, that are a subset of all the short memory processes

APPENDIX
D

Patterns to identify automated files

The patterns shown in this appendix have been in part obtained from the SlocCount, that identifies automated files using only the first 15 lines of the file. I have added some patterns, and extended the number of lines to 50, because after manual inspection I found some files that SlocCount could not identify as automated, but that were clearly the output of other tools. The lookup was case insensitive.

1. `generated automatically`
2. `automatically generated`
3. `generated by`
4. `a lexical scanner generated by flex`
5. `this is a generated file`
6. `generated with the.*utility`
7. `do not edit`
8. `autogenerated`
9. `machine generated`
10. `produced by`
11. `automatically written`
12. `created automatically`
13. `automatically created`
14. `codepage for`

15. `mapping table`
16. `generated from`
17. `conversion table`
18. `generated with`
19. `scanner table`

APPENDIX

E

Resumen en español

E.1 Antecedentes

Esta tesis consiste en el estudio empírico de una muestra muy grande de proyectos de software libre, con el fin de estudiar sus propiedades y su evolución.

La evolución de software es un campo que comenzó hace 40 años, con el trabajo de Meir M. Lehman. En 1969, mientras Lehman trabajaba en IBM, estudió los procesos de desarrollo dentro de la compañía, y extrajo algunas conclusiones usando diversas medidas tomadas de proyectos internos en IBM. Aunque inicialmente el informe era confidencial, los resultados acabaron publicándose en 1985, en el primer libro dedicado específicamente a la evolución de software [LB85]. En ese libro, Lehman enunció las *leyes de evolución de software*,

El estudio de Lehman se inició porque en aquella época existía preocupación acerca del rendimiento de los proyectos de desarrollo de software. Muchas veces esos proyectos no terminaban en el plazo estimado, y excedían los costes inicialmente previstos.

Para evitar estos problemas, Lehman hizo varias sugerencias, algunas de ellas resumidas en sus leyes. Sin embargo, los entornos donde se desarrolla software han cambiado mucho desde aquella época. Aunque Lehman ha cambiado el texto de las leyes en varias ocasiones para mantener su validez con las nuevas prácticas de desarrollo de software, en los últimos años han ido apareciendo casos donde esas leyes no se validan.

En particular, la aparición del fenómeno del software libre ha supuesto un modo de desarrollar software y gestionar proyectos de software que difiere mucho de las prácticas tradicionales usadas en la industria.

Los proyectos de software libre suelen distribuir versiones de los programas tan pronto como sea posible, y muchas veces se ofrecen al público con una mínima funcionalidad. El objetivo es mostrar un prototipo de lo que el proyecto intenta desarrollar, para ganar momento y atraer a nuevos desarrolladores al proyecto. Esta estrategia se conoce como *release early, release often*, y fue explicado por primera vez por Eric S. Raymond [Ray98].

Esta estrategia es diferente en proyectos de software en la industria, donde el software no se distribuye hasta que no esté listo y comprobado, y una vez puesto a disposición del público, los cambios en la funcionalidad son mínimos y las tareas de desarrollo son principalmente de mantenimiento.

Quizás por este motivo han aparecido artículos de investigación señalando que las leyes de Lehman no son válidas para algunos proyectos de software libre. El primero de estos artículos fue escrito por Godfrey y Tu [GT00]. En ese artículo se estudiaba el caso del kernel de Linux, y se hallaba que su curva de crecimiento era *súper-lineal*. En otras palabras, su crecimiento se estaba acelerando con el tiempo. Esto contraviene las leyes de Lehman, porque esas leyes dicen que el crecimiento de un proyecto de software se frena debido al incremento en la complejidad.

Este trabajo estimuló la publicación de artículos similares, que intentaban invalidar las leyes de Lehman. En cualquier caso, en nuestra opinión ésa no es la cuestión fundamental a la hora de estudiar la evolución del software. Es obvio que para un entorno completamente diferente al que Lehman tenía acceso para sus estudios, el resultado de aplicar los mismos métodos serán diferentes. Lehman intentaba buscar evidencias empíricas para justificar una teoría de evolución de software, y sus leyes condensaron cierto conocimiento que había sido validado en varios casos diferentes. Por tanto, en nuestra opinión, la verdadera cuestión que debe guiar a los estudios de evolución es si es posible obtener una teoría de evolución de software que sea universal, y que esté basada en evidencias científicas. Ése es el propósito de esta tesis.

En esta tesis nos ceñiremos al caso de software libre. El motivo principal es práctico. Existen cientos de miles de proyectos de software libre. La mayoría de ellos ofrecen sus repositorios (sistemas de control de versiones, código fuente, listas de correo, sistemas de seguimiento de fallos) disponibles públicamente, lo que permite que se puedan estudiar e incluso en ocasiones replicar.

Este último punto es muy importante, ya que permite replicar los resultados por terceros, de manera independiente. Es habitual en los artículos de investigación sobre Ingeniería del Software que los datos usados en el artículo se mantienen bajo secreto, imposibilitando la verificación imparcial por parte de terceros. Si pretendemos buscar las bases de una teoría de evolución de software, no podemos hacerlo sobre datos que se mantienen en secreto y sobre resultados que no se pueden verificar y validar (o invalidar).

E.2 Objetivos

El objetivo principal de esta tesis es estudiar una muestra masiva de proyectos de software libre, midiendo algunas de sus propiedades (tamaño, complejidad) y las características de su evolución.

Para acometer esta tarea hemos empleado un enfoque estadístico. De otro modo, dada la magnitud de la información manejada en esta tesis, no hubiera sido posible llevar a cabo los estudios que se describen en las próximas secciones.

Por tanto, podemos resumir el objetivo principal de esta tesis en el siguiente párrafo:

> Estudiar la evolución y propiedades de una muestra masiva de proyectos de software libre, usando un enfoque empírico y estadístico, para buscar patrones que pudieran usarse como base para enunciar una teoría universal de la evolución del software.

Entre las contribuciones principales de esta tesis, nosotros destacamos las dos siguientes:

- Es un estudio empírico realizado en una escala masiva, usando fuentes de datos públicamente disponibles, y bases de datos de investigación (FLOSSMole, CVSAnalY dataset). Este hecho facilita la verificación de este trabajo por terceros. Además, todas las bases de datos y *scripts* usados en esta tesis se han distribuido de manera pública, con el fin de facilitar estas verificaciones. Se pueden obtener en http://gsyc.es/~herraiz/phd/.
- Usa un enfoque estadístico para estudiar las propiedades y evolución del software. En particular, aplica análisis de series temporales para el estudio de la evolución de software.

Estas dos contribuciones han hecho posible por primera vez la visión original de Lehman [Leh74] de que la evolución de software debería ser estudiada de manera empírica, usando tantos casos de estudio como fuera posible, y empleando los métodos estadísticos adecuados (en particular, análisis de series temporales).

E.3 Metodología

La metodología empleada en esta tesis se divide en dos líneas principales:

- Estudio de las propiedades del software
- Estudio de la evolución del software

En cada una de las líneas, la metodología se divide en dos partes: recolección de datos, y análisis. Detallamos a continuación cada parte para cada una de las dos líneas.

E.3.1 Estudio de las propiedades del software

Recolección de datos

Para este estudio usamos una muestra de $12,108$ paquetes de software incluidos en el sistema operativo FreeBSD. Este sistema operativo contiene miles de paquetes de software, que son mantenidos por desarrolladores del proyecto. Se encargan de gestionar las dependencias, asegurarse que el software se puede compilar en el sistema, etc.

El sistema de paquetes contiene meta-información, que se puede emplear para obtener datos sobre miles de sistemas de software. En particular, nosotros usamos el sistema FreeBSD para obtener el código fuente de todos los paquetes, medir sus propiedades (tamaño, complejidad), y clasificarlos por campo de aplicación (gracias al atributo `section` incluido en los paquetes de FreeBSD).

De todos los paquetes, sólo medimos complejidad para los programas escritos en C. El lenguaje de programación C es el mayoritario en el mundo del software libre (el 51% del código en FreeBSD estaba escrito en C, y los porcentajes son similares en otras distribuciones de software libre [RGBM$^+$08]). Por tanto, esa muestra es suficientemente representativa.

Por tanto, para los paquetes escritos en C teníamos métricas de tamaño y complejidad, y sólo métricas de tamaño para el resto. Tras diversos procesos de filtrado, el conjunto contenía más de 1 millón de ficheros, de los que $447,000$ estaban escritos en C.

Análisis

Usando esa muestra de ficheros, realizamos dos análisis principales: calcular las correlaciones entre las métricas, y calcular las distribuciones estadísticas de las diferentes métricas.

Como cada fichero es generado de manera independiente, nosotros usamos los valores de las diferentes métricas de cada fichero como puntos para las correlaciones. Así, calculamos las diferentes correlaciones entre todas las métricas, a nivel de fichero.

Estas correlaciones se realizaron usando la muestra completa de ficheros. Sin embargo, esta muestra es heterogénea. Por ejemplo, contiene tanto ficheros de cabecera como ficheros de código fuente. Así que repetimos el análisis, dividiendo la muestra en grupos más pequeños y homogéneos. Usamos tres criterios para obtener grupos homogéneos:

- Tamaño del paquete
- Tipo de fichero (cabecera, código fuente)
- Campo de aplicación

Luego repetimos las correlaciones usando sólo cada uno de estos grupos que habíamos obtenido.

En cuanto a las distribuciones de tamaño, calculamos las distribuciones para la muestra general, y para cada uno de los grupos anteriores. Además, calculamos también las distribuciones a nivel de paquete, y a nivel de campo de aplicación. La métrica usada a nivel de paquete fue el número de ficheros contenido en cada paquete, y la métrica a nivel de campo de aplicación fue de número de paquetes incluido en cada campo de aplicación. Luego, intentamos caracterizar a qué distribución teórica correspondían las distribuciones obtenidas.

E.3.2 Estudio de la evolución del software

Recolección de datos

Para este análisis, se empleó una muestra de $3,821$ proyectos de software, obtenidos de las bases de datos de FLOSSMole [HCC06] y CVSAnalY dataset[1].

De todos los proyectos contenidos en esas bases de datos, nosotros seleccionamos los que cumplían con los siguientes criterios:

- Tener al menos 3 desarrolladores registrados.
- Tener al menos 1 año de historia en el sistema de control de versiones.
- Tener un repositorio de control de versiones.

La razón es que busćabamos proyectos con historia suficiente como para estudiar su evolución, y a la vez intentábamos descartar proyectos pequeños y abandonados.

Sobre el repositorio de control de versiones, era fundamental porque de lo contrario no hubiéramos podido obtener los datos necesarios para el análisis. La base de datos de CVSAnalY sólo contenía proyectos con repositorios CVS.

Una vez que seleccionamos esos proyectos, usando la base de datos de CVSAnalY, medimos el número diario de cambios ocurrido en cada uno de los proyectos. Esto se hizo usando dos métricas:

- Número de *commits*, tal y como los devuelve el sistema de control de versiones CVS.
- Número de *modification requests*, usando el algoritmo propuesto por Daniel M. German [Ger04].

La diferencia entre las dos métricas estriba en el modo de funcionamiento de CVS. Cada vez que se necesita hacer un cambio, ese cambio puede afectar a varios ficheros. Sin embargo, una vez que el cambio se registra en el CVS, cada fichero se registra por separado, y en vez de un único cambio aparecen varios cambios en el sistema. Para comprobar si existía alguna diferencia entre usar un parámetro u otro, nosotros decidimos medir los dos.

Análisis

Una vez obtenidos esos datos, intentamos caracterizar el perfil evolutivo de esos proyectos usando análisis de series temporales. En particular, calculamos la función de autocorrelación para cada proyecto, y calculamos si esa función correspondía a un proceso de memoria corta o de memoria larga. Esto se repitió con las dos métricas mencionadas anteriormente.

[1]Disponible de manera pública en http://libresoft.es/Results/CVSAnalY_SF

Una vez obtenida la función de autocorrelación, usamos modelos ARIMA (Auto-Regresivos, Integrados, y de Media movible) para predecir la evolución de tres casos de estudio.

E.4 Conclusiones

Una vez realizados todos los análisis explicados en la sección anterior, nosotros obtuvimos los siguientes resultados:

- Todas las métricas de tamaño y complejidad resultaron estar altamente correlacionadas, con las siguientes excepciones:
  - Ficheros de cabecera. En este caso, la complejidad ciclomática no está relacionada con ninguna otra métrica.
  - Nivel de Halstead. Esta métrica mostró correlaciones pobres cuando se repitió la correlación con grupos más homogéneos.
- Todas las métricas de tamaño y complejidad siguen una distribución de doble Pareto.
- La misma distribución de tamaño aparece a las escalas de paquete y campo de aplicación.
- La dinámica de la evolución de software es un proceso de memoria corta (del orden de días). Algunos proyectos tienen memorias largas, pero se les puede considerar marginales respecto a la mayoría de proyectos estudiados.
- Debido al punto anterior, los modelos ARIMA muestran un poder predictivo mucho mayor que modelos de regresión.

Estos resultados tiene varias implicaciones:

- Para medir las propiedades de un programa escrito en C, basta con medir el número de líneas de código, y el nivel de Halstead.
- Como la distribución de tamaño es doble Pareto, se podría emplear el *Random Forest File Model* [Mit04b] para simular la evolución de un proyecto de software.
- Una teoría de evolución de software debería tener en cuenta que la forma de la distribución de tamaño es la misma a diferentes escalas.
- Además, debería tener en cuenta que los eventos que ocurren hoy en un proyecto de software están influenciados por los eventos recientes, y que los eventos que ocurrieron hace mucho tiempo (más de varias semanas), no tiene ya ninguna influencia en el proyecto.

Como trabajo futuro de esta tesis queda validar la idoneidad de usar el *Random Forest File Model* adaptado al caso de evolución de software, y extender este estudio a otros casos de estudio, con el fin de comprobar si las distribuciones de tamaño son las mismas, y si la dinámica es también de memoria corta. De verificarse en muchos más casos de estudio, podrían ser los patrones necesarios para enunciar una teoría universal de evolución de software.

BIBLIOGRAPHY

[ACPM01] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *Proceedings of the International Conference on Software Maintenance*, pages 273–280, Florence, Italy, 2001. IEEE Computer Society.

[AH06] Mina Askari and Ric Holt. Information theoretic evaluation of change prediction models for large-scale software. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 126–132, New York, NY, USA, 2006. ACM.

[Aoy02] Mikio Aoyama. Metrics and analysis of software architecture evolution with discontinuity. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–107, New York, NY, USA, 2002. ACM.

[AP01] Annie I. Antón and Colin Potts. Functional paleontology: system evolution as the user sees it. In *Proceedings of the International Conference on Software Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society.

[AP03] Annie I. Antón and Colin Potts. Functional paleontology: The evolution of user-visible system services. *IEEE Transactions on Software Engeneering*, 29(2):151–166, 2003.

[ASAB02] I.P. Antoniades, I. Stamelos, L. Angelis, and GL Bleris. A novel simulation model for the development process of open source software projects. *Software Process Improvement and Practice*, 7(3-4):173–188, 2002.

[Asp93] William Aspray. Meir M. Lehman, Electrical Engineer, an oral history. IEEE History Center, 1993. Rutgers University, New Brunswick, NJ, USA.

[ASS$^+$04] I.P. Antoniades, I. Samoladas, I. Stamelos, L. Aggelis, and G. L. Bleris. Dynamical simulation models of the open source development process. In Stefan Koch, editor, *Free/Open Source Software Development*, pages 174–202. Idea Group Publishing, Hershey, PA, 2004.

[BDA$^+$99] P. Bourque, R. Dupuis, A. Abran, JW Moore, and L. Tripp. The guide to the Software Engineering Body of Knowledge. *IEEE Software*, 16(6):35–44, 1999.

[Bel79] L.A . Belady. On software complexity. In *Proceedings of the Workshop on Quantitative Software Models for Reliability*, pages 90–94, Kiamesha Lake, NY, USA, 1979. IEEE Computer Society.

[Bel85] L. A. Belady. *Program Evolution. Processes of Software Change.*, chapter On software complexity, pages 331–338. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[BFN$^+$06] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of java software. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 397–412, New York, NY, USA, 2006. ACM.

[BKS07] E.J. Barry, C.F. Kemerer, and S.A. Slaughter. How software process automation affects software evolution: a longitudinal empirical analysis. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(1):1–31, 2007.

[BL71] L. A. Belady and M. M. Lehman. Programming system dynamics or the metadynamics of systems in maintenance and growth. *Research Report RC3546, IBM*, 1971.

[BL76] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[BL85] L.A. Belady and M. M. Lehman. *Program Evolution. Processes of Software Change.*, chapter Programming system dynamics or the meta-dynamics of systems in maintenance and growth, pages 99–122. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[BT79] G. Benyon-Tinker. Complexity measures in an evolving large system. In *Proceedings of the Workshop on Quantitative Software Models for Reliability*, pages 117–127, Kiamesha Lake, NY, USA, 1979. IEEE Computer Society.

[BTW88] P. Bak, C. Tang, and K. Wiesenfeld. Self-organized criticality. *Physical Review A*, 38(1):364–374, 1988.

[CCPV01] Francesco Caprio, Gerardo Casazza, Massimiliano Di Penta, and Umberto Villano. Measuring and predicting the Linux kernel evolution. In *Proceedings of the International Workshop of Empirical Studies on Software Maintenance*, Florence, Italy, 2001.

[CFR05] A. Capiluppi, A.E. Faria, and J.F. Ramil. Exploring the relationship between cumulative change and complexity in an Open Source system. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 21–29. IEEE Computer Society, 2005.

[CFR07] A. Capiluppi and J. Fernandez-Ramil. A model to predict anti-regressive effort in Open Source Software. In *Proceedings of the International Conference on Software Maintenance*, pages 194–203. IEEE Computer Society, 2007.

[CG77] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–87, 1977.

[CGBHR07] Andrea Capiluppi, Jesús M. González-Barahona, Israel Herraiz, and Gregorio Robles. Adapting the "staged model for software evolution" to free/libre/open source software. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, pages 79–82, New York, NY, USA, 2007. ACM.

[CHLW06] Stephen Cook, Rachel Harrison, Meir M. Lehman, and Paul Wernick. Evolution in software systems: foundations of the SPE classification scheme. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):1–35, 2006.

[Cho80] C.K.S. Chong Hok Yuen. *A Phenomenology of Program Maintenance and Evolution*. PhD thesis, Imperial College, London, 1980.

[Cho85] C.K.S. Chong Hok Yuen. An empirical approach to the study of errors in large software under maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 96–105. IEEE Computer Society, 1985.

[Cho87] C.K.S. Chong Hok Yuen. A statistical rationale for evolution dynamics concepts. In *Proceedings of the International Conference on Software Maintenance*, pages 156–164. IEEE Computer Society, 1987.

[Cho88] C.K.S. Chong Hok Yuen. On analyzing maintenance process data at the global and detailed levels. In *Proceedings of the International Conference on Software Maintenance*, pages 248–255, Atlanta, GA, USA, 1988. IEEE Computer Society.

[Cle81] William S. Cleveland. LOWESS: A program for smoothing scatterplots by robust locally weighted regression. *The American Statistician*, 35(1):54, February 1981.

[CLM03] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. Characteristics of Open Source projects. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 317–327. IEEE Computer Society, 2003.

[CM07] Andrea Capiluppi and Martin Michlmayr. *Open Source development, adoption and innovation*, chapter From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects, pages 31–44. IFIP: International Federation for Information Processing. Springer Boston, 2007.

[CMPS07] Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708, 2007.

[CMR04a] Andrea Capiluppi, Maurizio Morisio, and Juan F. Ramil. The evolution of source folder structure in actively evolved Open Source systems. In *Proceedings of the International Symposium on Software Metrics*, pages 2–13. IEEE Computer Society, 2004.

[CMR04b] Andrea Capiluppi, Maurizio Morisio, and Juan F. Ramil. Structural evolution of an Open Source system: a case study. In *Proceedings of the International Workshop on Program Comprehension*, pages 172–183, Bari, Italy, 2004. IEEE Computer Society.

[CMS07] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. *Software Practice and Experience*, 37(6):581–641, 2007.

[Con86] Samuel D. Conte. *Software Engineering Metrics and Models (Benjamin/Cummings series in software engineering)*. Benjamin-Cummings Pub Co, 1986.

[CR04] Andrea Capiluppi and Juan F. Ramil. Studying the evolution of Open Source systems at different levels of granularity: two case studies. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 113–118. IEEE Computer Society, 2004.

[CSN07] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data, 2007.

[DD03] Jean-Michel Dalle and Paul A. David. The allocation of software development resources in Open Source production mode. Technical report, SIEPR Policy paper No. 02-027, SIEPR, Stanford, USA, 2003. http://siepr.stanford.edu/papers/pdf/02-27.pdf.

[DDdB06] Jean-Michell Dalle, Laurent Daudet, and Matthisj den Besten. Mining CVS signals. In *Proceedings of the Workshop on Public Data about Software Development*, pages 12–21, Como, Italy, 2006.

[Dow01] Allen B. Downey. The structural cause of file size distributions. In *Proceedings of International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 361–370, Cincinnati, OH, USA, 2001. IEEE Computer Society.

[Dow05] Allen B. Downey. Lognormal and Pareto distributions in the Internet. *Computer Communications*, 28(7):709–801, 2005.

[Dvo94] Joseph Dvorak. Conceptual entropy and its effect on class hierarchies. *IEEE Computer*, 27(6):59–63, 1994.

[EGWEH02] Letha H. Etzkorn, Sampson Gholston, and Jr. William E. Hughes. A semantic entropy metric. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(4):293–310, 2002.

[FB02] Eduardo Fuentetaja and Donald J. Bagert. Software Evolution from a Time-Series perspective. In *Proceedings of the International Conference on Software Maintenance*, pages 226–229. IEEE Computer Society, 2002.

[FKP02] Alex Fabrikant, Elias Koutsoupias, and Christos H. Papadimitriou. Heuristically optimized trade-offs: A new paradigm for power laws in the Internet. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, pages 110–122, London, UK, 2002. Springer-Verlag.

[FNPAQ00] Anna R. Fasolino, Domenico Natale, Alessio Poli, and Alessandro Alberigi-Quaranta. Metrics in the development and maintenance of software: an application in a large scale environment. *Journal of Software Maintence: Research and Practice*, 12:343–355, 2000.

[FRLWC08] Juan Fernandez-Ramil, Angela Lozano, Michel Wermelinger, and Andrea Capiluppi. *Software Evolution*, chapter Empirical Studies of Open Source Evolution, pages 263–288. Springer, 2008.

[GB04] Jesus M. Gonzalez-Barahona. Quo vadis, libre software?, 2004. http://sinetgy.org/~jgb/articulos/libre-software-origin/.

[Ger04] Daniel M. German. Mining CVS repositories, the softchange experience. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 17–21, Edinburg, Scotland, UK, 2004.

[GH06] Daniel M. German and Abraham Hindle. Visualizing the evolution of software using softChange. *International Journal of Software Engineering and Knowledge Engineering*, 16(1):5–21, 2006.

[GJKT97] Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance*, pages 160–170. IEEE Computer Society, 1997.

[GT00] Michael W. Godfrey and Quiang Tu. Evolution in Open Source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142, San Jose, California, 2000.

[GT01] Michael Godfrey and Qiang Tu. Growth, evolution, and structural change in open source software. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–106, Vienna, Austria, 2001.

[Har92] Warren Harrison. An entropy-based measure of software complexity. *IEEE Transactions on Software Engineering*, 18(11):1025–1029, 1992.

[HCC06] James Howison, Megan Conklin, and Kevin Crowston. FLOSSMole: a collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, July-September 2006.

[HGBR07a] Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Forecasting the number of changes in Eclipse using time series analysis. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 32–33, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[HGBR07b] Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a theoretical model for software growth. In *International Workshop on Mining Software Repositories*, pages 21–30, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[HRGB$^+$06] Israel Herraiz, Gregorio Robles, Jesus M. Gonzalez-Barahona, Andrea Capiluppi, and Juan F. Ramil. Comparison between SLOCs and number of files as size metrics for software evolution analysis. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 203–210, Bari, Italy, 2006. IEEE Computer Society.

[HWH05] Ahmed E. Hassan, Jingwei Wu, and Richard C. Holt. Visualizing historical data using spectrographs. In *Proceedings of the International Symposium on Software Metrics*, Como, Italy, 2005. IEEE Computer Society.

[Kan03] Stephen H. Kan. *Metrics and Models in Software Quality Engineering (2nd Edition)*. Addison-Wesley Professional, September 2003.

[KCM07] H. Kagdi, M.L. Collard, and J.I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[Kit82] B. Kitchenham. System evolution dynamics of VME/B. *ICL Technical Journal*, 3:43–57, 1982.

[Knu71] Donald E. Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, 1(2):105–133, 1971.

[Koc05] Stefan Koch. Evolution of Open Source Software systems - a large-scale investigation. In *Proceedings of the International Conference on Open Source Systems*, Genova, Italy, July 2005.

[KS99] C. F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, 1999.

[KSW95] K. Kim, Y. Shin, and C. Wu. Complexity measures for object-oriented program based on the entropy. In *Proceedings of the Asia Pacific Software Engineering Conference*, pages 127–135, Washington, DC, USA, 1995. IEEE Computer Society.

[Law82] M. J. Lawrence. An examination of evolution dynamics. In *Proceedings of the International Conference on Software Engineering*, pages 188–196, Tokyo, Japan, 1982. IEEE Computer Society Press.

[LB85] M. M. Lehman and L. A. Belady, editors. *Program evolution. Processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Leh74] M. M. Lehman. Programs, Cities, Students: Limits to Growth?, 1974. Inaugural lecture, Imperial College of Science and Technology, University of London.

[Leh78] M. M. Lehman. Laws of Program Evolution-Rules and Tools for Programming Management. In *Proceedings of Infotech State of the Art Conference, Why Software Projects Fail*, 1978.

[Leh80] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[Leh85a] M. M. Lehman. *Program Evolution. Processes of Software Change.*, chapter The Programming Process, pages 39–84. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Leh85b] M. M. Lehman. *Program Evolution. Processes of Software Change.*, chapter Programs, Cities, Students: Limits to Growth?, pages 133–164. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Leh85c] M. M. Lehman. *Program Evolution. Processes of Software Change.*, chapter Laws of Program Evolution-Rules and Tools for Programming Management, pages 247–274. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Leh96] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag.

[LP76] M. M. Lehman and F. N. Parr. Program Evolution and its impact on Software Engineering. In *Proceedings of the International Conference on Software Engineering*, pages 350–357, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[LP85] M. M. Lehman and F. N. Parr. *Program Evolution. Processes of Software Change.*, chapter Program Evolution and its impact on Software Engineering, pages 201–220. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[LPR98] Manny M. Lehman, Dewayne E. Perry, and Juan F. Ramil. Implications of evolution metrics on software maintenance. In *Proceedings of International Conference on Software Maintenance*, pages 208–217. IEEE Computer Society, 1998.

[LR02] M.M. Lehman and J.F. Ramil. An overview of some lessons learnt in FEAST. In *Proceedings of the Workshop on Empirical Studies of Software Maintenance*, 2002.

[LRS01] Manny M. Lehman, Juan F. Ramil, and U. Sandler. An approach to modelling long-term growth trends in software systems. In *Internation Conference on Software Maintenance*, pages 219–228, Florence, Italy, 2001. IEEE Computer Society.

[LRW$^+$97] Manny M. Lehman, Juan F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the International Symposium on Software Metrics*, 1997.

[LSV08] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 2008. To appear.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[MFRP06] Nazim H. Madhavji, Juan Fernandez-Ramil, and Dewayne E. Perry, editors. *Software Evolution and Feedback. Theory and Practice.* Wiley, 2006.

[Mit04a] Michael Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2004.

[Mit04b] Michael Mitzenmacher. Dynamic models for file sizes and double Pareto distributions. *Internet Mathematics*, 1(3):305–333, 2004.

[Mit05] Michael Mitzenmacher. Editorial: The future of power law research. *Internet Mathematics*, 2(4):525–534, 2005.

[MWH98] Spyros G. Makridakis, Steven C. Wheelwright, and Rob J. Hyndman. *Forecasting: Methods and Applications*. John Wiley & Sons, Ltd., January 1998.

[NR69] Peter Naur and Brian Randell, editors. *Software Engineering. Report on a conference sponsored by the NATO Scientific Committee*, Brussels, January 1969. NATO Scientific Committee. Conference celebrated in 1968. Proceedings published as a report in 1969.

[Pen99] Roger Penrose. *The Emperor's New Mind*. Oxford University Press, 1999.

[Per94] Dewayne E. Perry. Dimensions of software evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 296–303, Washington, DC, USA, 1994. IEEE Computer Society.

[Per99] Bruce Perens. The open source definition. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly and Associates, Cambridge, Massachusetts, 1999.

[Per06] Dewayne E. Perry. *Software Evolution and Feedback. Theory and Practice*, chapter A nontraditional view of the dimensions of software evolution, pages 41–51. Wiley, 2006.

[Pir88] Shamin S. Pirzada. *A staistical examination of the evolution of the UNIX system.* PhD thesis, Imperial College. University of London., 1988.

[PLM07] Yi Peng, Fu Li, and Ali Mili. Modeling the evolution of operating systems: An empirical study. *Journal of Systems and Software*, 80(1):1–15, 2007.

[PSE04] J. W. Paulson, Giancarlo Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering*, 30(4), April 2004.

[RAGBH05] Gregorio Robles, Juan Jose Amor, Jesus M. Gonzalez-Barahona, and Israel Herraiz. Evolution and growth in large libre software projects. In *Proceedings of the International Workshop on Principles in Software Evolution*, pages 165–174, Lisbon, Portugal, September 2005.

[Ran69] Brian Randell. Towards a methodology of computing system design. In Peter Naur and Brian Randell, editors, *Software Engineering*, pages 204–208, Garlisch, Germany, 1969. NATO Scientific Committee. Conference celebrated in 1968. Proceedings published as a report in 1969.

[Ray98] Eric S. Raymond. The cathedral and the bazaar. *First Monday*, 3(3), March 1998.
http://www.firstmonday.dk/issues/issue3_3/raymond/.

[RB00] V.T. Rajlich and K.H. Bennett. A staged model for the software life cycle. *IEEE Computer*, 33(7):66–71, Jul 2000.

[RGBM$^+$08] Gregorio Robles, Jesus M. Gonzalez-Barahona, Martin Michlmayr, Juan Jose Amor, and Daniel M. German. Macro-level software evolution: A case study of a large software compilation. *Empirical Software Engineering*, 2008. To appear.

[RJ04] W.J. Reed and M. Jorgensen. The Double Pareto-Lognormal Distribution. A New Parametric Model for Size Distributions. *Communications in Statistics-Theory and Methods*, 33(8):1733–1753, 2004.

[RL00] Juan F. Ramil and Meir M. Lehman. Metrics of software evolution as effort predictors - a case study. In *Proceedings of the International Conference on Software Maintenance*, pages 163–172. IEEE Computer Society, 2000.

[RMGB05] Gregorio Robles, Juan Julian Merelo, and Jesus M. Gonzalez-Barahona. Self-organized development in libre software: a model based on the stigmergy concept. In *Proceedings of the International Workshop on Software Process Simulation and Modeling*, St.Louis, Missouri, USA, 2005.

[Rob06] Gregorio Robles. *Empirical Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. PhD thesis, Universidad Rey Juan Carlos, 2006.

[Roc96] Jose Luis Roca. An entropy-based method for computing software structural complexity. *Microelectronics and Reliability*, 36(5):609–620, May 1996.

[Som06] I. Sommerville. *Software Engineering*. Addison Wesley Publishing Company, 2006.

[SS06] Robert H. Shumway and David S. Stoffer. *Time Series Analysis and Applications. With R Examples*. Springer Texts in Statistics. Springer, 2006.

[Sta02] R.M. Stallman. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Gnu Press, 2002.

[Tur96] Wladyslaw M. Turski. Reference model for smooth growth of software systems. *IEEE Transactions on Software Engineering*, 22(8):599–600, 1996.

[Tur02] W. M. Turski. The reference model for smooth growth of software systems revisited. *IEEE Transactions on Software Engineering*, 28(8):814–815, 2002.

[WHH07] Jingwei Wu, Richard Holt, and Ahmed E. Hassan. Empirical evidence for SOC dynamics in software evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 244–254. IEEE Computer Society, 2007.

[Woo80] C. Murray Woodside. A mathematical model for the evolution of software. *Journal of Systems and Software*, 1:337–345, 1980.

[Wu06] Jingwei Wu. *Open Source Software evolution and its dynamics*. PhD thesis, University of Waterloo, 2006.